The Pragmatic Programmers

TURING 图灵程序设计丛书

Erlang之父权威著作

带你领先一步，精通下一代主流编程语言

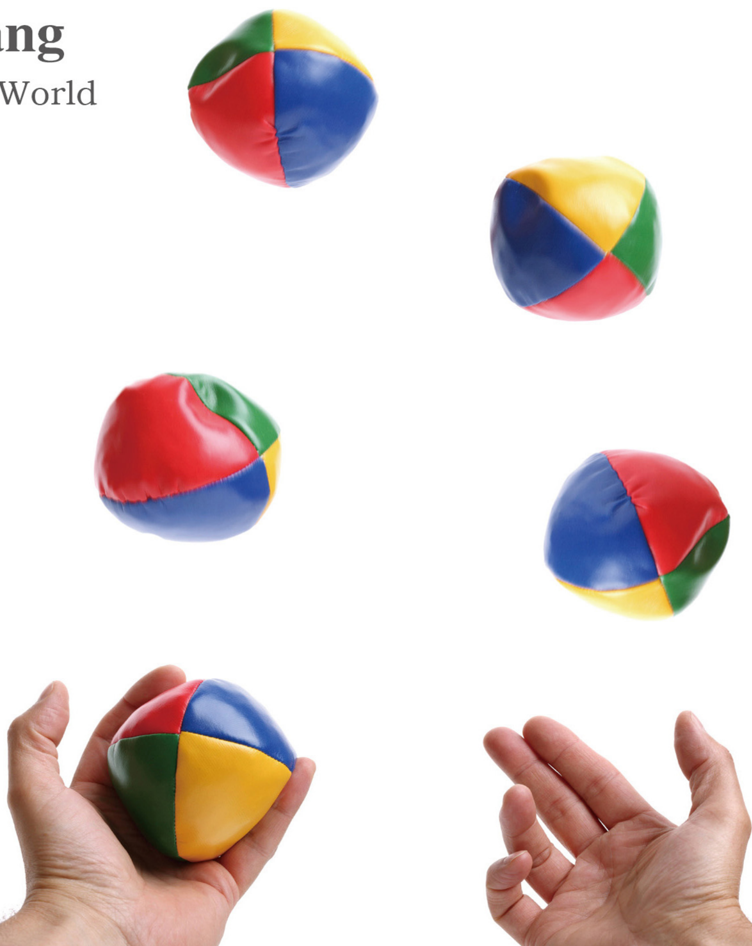
Erlang程序设计

（第2版）

Programming Erlang

Software for a Concurrent World

Second Edition



【瑞典】Joe Armstrong 著

牛化成 译

人民邮电出版社
POSTS & TELECOM PRESS

“Erlang是目前唯一成熟可靠的能够开发高扩展性并发软件系统的语言，它将成为下一个Java。”

——Ralph Johnson，
软件开发大师，《设计模式》作者之一

“Joe的《Erlang程序设计》一书影响巨大。第2版做了重要更新，万众期待，不但涵盖核心语言和框架的基本内容，还涉及rebar和cowboy这样的关键社区项目。有经验的Erlang程序员也能在书里找到各种有用的提示和新见解，初学者则会喜欢Joe在介绍和阐释关键语言概念时所使用的清楚和有条理的方式。”

——Alexander Gounares，
AOL前CTO，比尔•盖茨的顾问，
Concurix公司的创始人兼CEO

“一部佳作。对函数式编程的介绍理性且实用。”

——Gilad Bracha，
Java语言和Java虚拟机规范的共同作者，
Newspeak语言的创造者，Dart语言团队成员

“本书是理解如何进行Actor编程的优秀资源，不仅适用于Erlang开发人员，对于那些想要理解Actor为何如此重要，以及为何它们是构建反应式、可扩展、可恢复和事件驱动型系统的重要工具的程序员，也同样适用。”

—— Jonas Boner，
Akka项目和AspectWerkz面向方面编程框架
创立者，Typesafe联合创始人兼CTO

“Erlang让我有醍醐灌顶之感，它促使我开始以完全不同的方式思考问题，Armstrong能够亲自写作本书，实乃Erlang爱好者之福。”

—— David Thomas，软件开发大师，
《程序员修炼之道》作者

图灵程序设计丛书

Erlang程序设计

（第2版）

【瑞典】Joe Armstrong 著
牛化成 译

Programming Erlang

Software for a Concurrent World

Second Edition

人 民 邮 电 出 版 社
北 京

图书在版编目（C I P）数据

Erlang程序设计 : 第2版 / (瑞典) 阿姆斯特朗
(Armstrong,J.) 著 ; 牛化成译. -- 北京 : 人民邮电出
版社, 2014.6
（图灵程序设计丛书）
ISBN 978-7-115-35457-0

Ⅰ. ①E… Ⅱ. ①阿… ②牛… Ⅲ. ①程序语言—程序
设计 Ⅳ. ①TP312

中国版本图书馆CIP数据核字(2014)第097736号

内 容 提 要

本书由 Erlang 之父 Joe Armstrong 编写，是毋庸置疑的经典著作。书中兼顾了顺序编程、并发编程和分布式编程，深入讨论了开发 Erlang 应用中至关重要的文件和网络编程、OTP、ETS 和 DETS 等主题。新版针对入门级程序员增加了相关内容。

本书适合 Erlang 初学者和中级水平 Erlang 程序员学习参考。

◆ 著　　　　[瑞典] Joe Armstrong
译　　　　牛化成
责任编辑　朱　巍
执行编辑　陈婷婷
责任印制　焦志炜

◆ 人民邮电出版社出版发行　　北京市丰台区成寿寺路11号
邮编　100164　　电子邮件　315@ptpress.com.cn
网址　http://www.ptpress.com.cn
北京　　　　　　印刷

◆ 开本：800×1000　1/16
印张：28
字数：662千字　　　　2014年 6 月第 1 版
印数：1－4 000册　　　2014年 6 月北京第 1 次印刷
著作权合同登记号　图字：01-2013-8803号

定价：89.00元

读者服务热线：(010)51095186转600　印装质量热线：(010)81055316

反盗版热线：(010)81055315

广告经营许可证：京崇工商广字第 0021 号

版 权 声 明

这个世界是并行的。

如果希望将程序的行为设计得与真实世界物体的行为相一致，那么程序就应该具有并发结构。

使用专门为并发应用设计的语言，开发将变得极为简便。

Erlang程序模拟了人类如何思考，如何交互。

——Joe Armstrong

第1版推荐序

Erlang算不上是一种“大众流行”的程序设计语言，而且即使是Erlang的支持者，大多数也对于Erlang成为“主流语言”并不持乐观态度。然而，自从2006年以来，Erlang语言确实在国内外一批精英程序员中暗流涌动，光我所认识和听说的，就有不少于一打技术高手像着了魔一样迷上了这种已经有二十多年历史的老牌语言。这是一件相当奇怪的事情。因为就年龄而言，Erlang大约与Perl同年，比C++年轻四岁，长Java差不多十岁，但Java早已经是工业主流语言，C++和Perl甚至已经进入其生命周期的下降阶段。照理说，一个被扔在角落里二十多载无人理睬的老家伙合理的命运就是坐以待毙，没想到Erlang却像是突然吃了返老还童丹似的在二十多岁的“高龄”又火了一把，不但对它感兴趣的人数量激增，而且还成立了一些组织，开发实施了一些非常有影响力的软件项目。这是怎么回事呢？

根本原因在于Erlang天赋异禀恰好适应了计算环境变革的大趋势：CPU的多核化与云计算。

自2005年C++标准委员会主席Herb Sutter在*Dr. Dobb's Journal*上发表《免费午餐已经结束》一门以来，人们已经确凿无疑地认识到，如果未来不能有效地以并行化的软件充分利用并行化的硬件资源，我们的计算效率就会永远停滞在仅仅略高于当前的水平上，而不得动弹。因此，未来的计算必然是并行的。Herb Sutter本人曾表示，如果一门语言不能够以优雅可靠的方式处理并行计算的问题，那它就失去了在21世纪的生存权。“主流语言”当然不想真的丧失掉这个生存权，于是纷纷以不同的方式解决并行计算的问题。就C/C++而言，除了标准委员会致力于以标准库的方式来提供并行计算库之外，标准化的OpenMP和MPI，以及Intel的Threading Building Blocks库也都是可信赖的解决方案；Java在5.0版中引入了意义重大的concurrency库，得到Java社区的一致推崇；而微软更是采用了多种手段来应对这一问题：先是在.NET中引入APM，随后又在Robotics Studio中提供了CCR库，最近又发布了Parrallel FX和MPI.NET，可谓不遗余力。然而，这些手法都可以视为亡羊补牢，因为这些语言和基础设施在创造时都没有把并行化的问题放到优先的位置来考虑。与它们相反，Erlang从其构思的时候起，就把“并行”放到了中心位置，其语言机制和细节的设计无不从并行角度出发和考虑，并且在长达二十年的发展完善中不断成熟。今天，Erlang可以说是为数不多的天然适应多核的可靠计算环境，这不能不说是一种历史的机缘。

另一个可能更加迫切的变革，就是云计算。Google的实践表明，用廉价服务器组成的服务器集群，在计算能力、可靠性等方面能够达到价格昂贵的大型计算机的水准，毫无疑问，这是大型、超大型网站和网络应用梦寐以求的境界。然而，要到达这个境界并不容易。目前一般的网站为了达成较好的可延展性和运行效率，需要聘请有经验的架构师和系统管理人员，手工配置网络服务

端架构，并且常备一个高水准的系统运维部门，随时准备处理各种意外情况。可以说，虽然大多数Web企业只不过是想在这些基础设施上运行应用而已，但仅仅为了让基础设施正常运转，企业就必须投入巨大的资源和精力。现在甚至可以说，这方面的能力成了大型和超大型网站的核心竞争力。这与操作系统成熟之前人们自己动手设置硬件并且编写驱动程序的情形类似——做应用的人要精通底层细节。这种格局的不合理性一望便知，而解决的思路也是一目了然——建立网络服务端计算的操作系统，也就是类似Google已经建立起来的"云计算"那样的平台。所谓"云计算"，指的是结果，而当前的关键不是这个结果，而是作为手段的"计算云"。计算云实际上就是控制大型网络服务器集群计算资源的操作系统，它不但可以自动将计算任务并行化，充分调动大型服务器集群的计算能力，而且还可以自动应对大多数系统故障，实现高水平的自主管理。计算云技术是网络计算时代的操作系统，是绝对的核心技术，也正因此，很多赫赫有名的中外大型IT企业都在不惜投入巨资研发计算云。包括我在内的很多人都相信，云计算将不仅从根本上改变我们的计算环境，而且将从根本上改变IT产业的盈利模式，是真正几十年一遇的重大变革，对于一些企业和技术人员来说是重大的历史机遇。恰恰在这个主题上，Erlang又具有先天的优势，这当然也是归结于其与生俱来的并行计算能力，使得开发计算云系统对于Erlang来说格外轻松容易。现在Erlang社区已经开发了一些在实践中被证明非常有效的云计算系统，学习Erlang和这些系统是迅速进入这个领域并且提高水平的捷径。

由此可见，Erlang虽然目前还不是主流语言，但是有可能会在未来一段时间发挥重要的作用，因此，对于那些愿意领略技术前沿风景的"先锋派"程序员来说，了解和学习Erlang可能是非常有价值的投资。即使你未来不打算使用Erlang，也非常有可能从Erlang的设计和Erlang社区的智慧中得到启发，从而能够在其他语言的项目中更好地完成并行计算和云计算相关的设计和实现任务。再退一步说，就算只是从开启思路、全面认识计算本质和并行计算特性的角度出发，Erlang也值得了解。所以，我很希望这本书在中国程序员社区中不要遭到冷遇。

本书是由Erlang创造者Joe Armstrong亲自执笔撰写的Erlang语言权威参考书，原作以轻松引导的方式帮助读者在实践中理解Erlang的深刻设计思路，并掌握以Erlang开发并行程序的技术，在技术图书中属于难得的佳作。两位译者我都认识，他们都是技术精湛而思想深刻的"先锋派"，对Erlang有着极高的热情，因此翻译质量相当高，阅读起来流畅通顺，为此书中译本添色不少。有兴趣的读者集中一段时间按图索骥，完全有可能就此踏上理解Erlang、应用Erlang的大路。

孟岩
IBM中国公司媒体关系主管
前CSDN首席分析师兼《程序员》杂志技术主编

前　　言

越来越多的新硬件是并行的，所以新的编程语言必须支持并发性，否则它们将会灭亡。

“处理器行业的发展方向是不断添加更多的核心，但是没人知道如何给它们编程。我的意思是：双核？可以。四核？很难。八核？算了吧。”

——史蒂夫·乔布斯[1]

乔布斯说得不对，我们其实知道如何给多核编程。我们在Erlang里就这么做，而且核心越多，许多程序就跑得越快。

Erlang从一开始就被设计用于自下而上地编写并发式、分布式、容错、可扩展和软实时（soft real-time）系统的程序。软实时系统是指电话交换机和银行业务系统这样的系统，对它们而言，快速的响应时间很重要，但偶尔错过了时限也不是什么灾难性的。Erlang系统已经被大规模部署，并且控制了全世界许多重要的移动通信网络。

如果你的问题是并发的，或者正在组建多用户的系统，或者你组建的系统需要随时间而改变，那么使用Erlang也许会为你节省大量的工作，因为Erlang就是特别为组建这些系统而设计的。

“问题在于可变状态，傻瓜。”

——摘自Brian Goetz所著《Java并发编程实战》

Erlang是函数式编程语言。函数式编程禁止代码存在副作用。副作用和并发性不能共存。在Erlang里，改变单个进程内部的状态是允许的，但一个进程改动另一个的状态则不行。Erlang里没有互斥，没有同步方法，也没有内存共享式编程的各种设备。

各个进程能且只能用一种方法进行交互，那就是交换消息。进程之间不共享任何数据。这就是我们可以把Erlang程序轻松部署到多核或网络的原因。

编写Erlang程序时，实现的方式不是让单个进程执行所有任务，而是生成大量只做简单事情的小进程，并让它们互相通信。

本书主要内容

本书介绍并发性、分布、容错和函数式编程，阐述如何编写一个没有锁与互斥、只是纯粹使

① http://bits.blogs.nytimes.com/2008/06/10/apple-in-parallel-turning-the-pc-world-upside-down/

用消息传递的分布式并发系统，如何在多核CPU上自动加速程序，如何编写让人们互相交流的分布式应用程序。还介绍了如何编写容错和分布式系统的设计模式，如何给并发性建模、再把这些模型映射到计算机程序上（我把这一过程称为*面向并发性的程序设计*）。

目标读者

本书的目标读者上至经验丰富但想要了解更多Erlang内部细节和背后哲学的Erlang程序员，下至不折不扣的初学者。书中的内容已经被从专家到初学者的各级程序员审读过。第2版与第1版的一个主要区别是，第2版添加了大量针对初学者的解释性材料。高级Erlang程序员可以跳过这些介绍材料。

本书还想阐明函数式编程、并发编程与分布式编程，并将它们以合适的方式呈现给之前不了解并发或函数式编程的读者。编写函数式程序和并行程序长久以来都被当成是一种“巫术”，希望本书能让人们对此有新的认识。

虽然本书不假定读者拥有特定的函数式或并发编程知识，但是需要读者熟悉一两种编程语言。

当你接触一门新的编程语言时，往往难以想到“适合用新语言解决的问题”。书中的练习会给你一些提示。这些问题很适合在Erlang里解决。

新版说明

首先，书中的内容已经做了更新，以反映出自第1版面世以来Erlang历经的所有变化。现在我们已经涵盖了所有的官方语言改动，并介绍了Erlang的R17版。

第2版把焦点转移到了满足初学者的需求上，相比第1版有了更多的解释性文字。那些针对高级用户或可能迅速变化的材料已经被转移到了在线资源区。

事实证明，第1版里的编程练习非常受读者欢迎，因此现在每一章的最后都附上了练习。这些练习复杂程度各异，所以初学者和高级用户总能找到适合自己的。

在新增的一些章节中，你将学到Erlang的类型系统和Dialyzer、映射组（在Erlang的R17版里新增）、WebSocket、编程习惯用语及如何集成第三方代码。还新增了一篇附录，介绍如何组建一个独立的最小化Erlang系统。

本书的最后一章“福尔摩斯的最后一案”是全新的，里面提供了一项练习：处理大批量的文本并从中提取意义。这一章是开放式的，希望它后面的练习能激发更进一步的工作。

路线图

要学会跑，必须先学会走路。Erlang程序是由许多同时运行的小型顺序程序组成的。在编写并发代码之前，我们需要能够编写顺序代码。这就意味着在第11章之前，我们不会深入到编写并发程序的细节当中。

- 第一部分简单介绍了并发编程的核心概念，并带你快速游历Erlang。

- 第二部分详细介绍了Erlang的顺序编程，还讨论了构建Erlang应用程序所需的类型和方法。
- 第三部分是本书的核心，我们将学习如何编写并发和分布式的Erlang程序。
- 第四部分讨论了主要的Erlang库、跟踪和调试的技巧，以及组织Erlang代码的技巧。
- 第五部分涉及应用程序。你会学到如何将外部软件与Erlang的核心库集成，以及如何转换你自己的代码来把它奉献给开源事业。我们将讨论编程习惯用语和如何为多核CPU编程。最后，“夏洛克·福尔摩斯”会分析我们的思路。

在每一章的最后，你都会看到一组精心挑选的编程练习。它们的目的是测试你对本章知识的掌握情况，同时向你发起挑战。这些问题难度各异，最大的难题是合适的研究项目。即使你并不打算尝试解决所有问题，单单思考问题本身以及如何解决问题也会增强你对书中内容的理解。

本书里的代码

大多数代码片段都来源于可下载的完整运行范例[1]。为了方便读者查询，如果书中的某个代码清单可供下载，代码片段上方就会有一个提示条（就像此处所显示的）：

shop1.erl

```
-module(shop1).
-export([total/1]).

total([{What, N}|T]) -> shop:cost(What) * N + total(T);
total([])            -> 0.
```

这个提示条内含代码在下载区里的路径。如果你正在阅读本书的电子版，并且你的电子书阅读器支持超链接，就可以点击提示条，代码应该会出现在浏览器窗口中。

帮帮我！出问题了

学习新东西并不容易，你会遇到棘手的难题。当你遇到难题时，原则一是不要轻易放弃。原则二是寻求帮助。原则三是询问“福尔摩斯”。

原则一很重要。有些人尝试了Erlang，遇到难题后选择了放弃，没有告诉任何人。如果我们不知道某个问题存在，我们就无法修复它。

寻求帮助的最佳方式是首先用谷歌找找看，如果谷歌帮不上忙，你可以给Erlang的邮件列表[2]发送邮件。如果想要更快速的响应，还可以试试`irc.freenode.net`上的`#erlounge`或`#erlang`。

有时候，问题的答案也许在Erlang邮件列表的某篇旧帖子上，但你就是找不到。在第27章有一个可以本地运行的程序，它能对Erlang邮件列表里的所有旧帖子执行复杂的搜索。

言归正传，下面感谢那些帮助我编写本书（以及本书第1版）的人们。你可以跳过去这一内容直奔第1章，在那里我将带你快速游历Erlang。

---

① http://www.pragprog.com/titles/jaerlang2/source_code

② erlang-questions@erlang.org

致谢

第 1 版

许多人对本书的准备工作提供了帮助，我想在这里对他们表示感谢。

首先，感谢编辑Dave Thomas。Dave指导我写作，提出的问题无穷无尽。为什么要这样？为什么要那样？当我开始编写这本书时，Dave说我的写作风格就像是“站在岩石上说教”。他说：“我想要你和人们对话，而不是说教。”这本书因此变得更好了。谢谢你，Dave！

其次，感谢我身后由语言专家组成的委员会。他们帮助我决定该省去哪些内容，还帮助我阐明了某些难以解释的部分。在这里感谢（排名不分先后）Björn Gustavsson、Robert Virding、Kostis Sagonas、Kenneth Lundin、Richard Carlsson和Ulf Wiger。

感谢Claes Vikström提供了有关Mnesia的宝贵建议，感谢Rickard Green提供了有关SMP Erlang的信息，也感谢Hans Nilsson提供了用于文本索引程序的单词归一算法（stemming algorithm）。

Sean Hinde和Ulf Wiger帮助我理解了如何使用各种OTP内部细节。Serge Aleynikov向我解释了活动套接字以帮助我理解。

感谢妻子Helen Taylor，她校对了好几章，并在我需要提神醒脑时端来热茶。不仅如此，她还容忍了我在7个月的时间里表现出的相当沉迷的行为。另外还要感谢Thomas和Claire，感谢Bach和Handel，感谢我的猫Zorro和Daisy，以及我的“卫星定位”Doris，它帮助我保持理智，在被抚摸时会呜呜叫，还会把我带往正确的地址。

最后，感谢所有填写了勘误请求的书稿读者：我咒骂过你们，也赞美过你们。当第一稿面世时，出乎我的意料，整本书在两天内就被读完，你们的评论把每一页都撕成了碎片。但是，这个过程促成了一本好书的诞生，它的质量远超我的想象。当许多人说“我看不懂这一页”时（这发生过好多次），我就被迫重新思考并改写相关内容。谢谢你们每个人的帮助。

第 2 版

首先，我的新编辑Susannah Pfalzer关于调整本书组织方式和重心的建议很有用。和你一起工作很棒，你教会了我很多。

Kenneth Lundin和OTP小组的成员们努力工作，实现了第2版里描述的那些新的语言特性。

第1版的许多读者反馈了他们所不理解的内容，我希望现在这些已经被修正了。

映射组（map）的设计灵感来源于Richard A. O’Keefe的工作（他把它们称为“框架”，即frame）。Richard多年来一直在Erlang邮件列表里捍卫框架的价值。谢谢你的评论和建议，Richard。

Kostis Sagonas在处理类型系统方面提供了许多有用的反馈。

我还想感谢Loïc Hoguin允许我们使用Nine Nines公司Cowboy Web服务器的一些代码作为例子，以及那些来自Basho、为BitLocker编写代码的人们。我还想感谢Dave Smith对rebar所做的工作。

许多人帮助我审读了不同批次的第2版草稿。我想感谢他们所有人，因为他们让这本书变得更好：Erik Abefelt、Paul Butcher、Mark Chu-Carroll、Ian Dees、Henning Diedrich、Jeremy Frens Loïc Hoguin、Andy Hunt、Kurt Landrus、Kenneth Lundin、Evan Miller、Patrik Nyblom、Tim Ottinger、Kim Shrier和Bruce Tate。

感谢Helen Taylor（Twitter @mrsjoeerl）为我提供无数杯热茶，在我几欲失去创作动力时给我鼓劲。

感谢古斯塔夫·马勒、拉赫曼尼诺夫、理查德·瓦格纳和乔治·弗里德里希·亨德尔（以及鲍勃·迪伦和其他一些人）创作了我编写本书大部分内容时播放的背景音乐。

目　　录

第五部分 构建应用程序

Part 1

第一部分

为何用 Erlang

本部分将介绍什么是并发，然后讨论并发和并行的区别。你将会了解到编写并发程序的益处，并快速游历 Erlang，其中介绍了这种语言的主要特性。

第 1 章

什么是并发

让我们暂时忘却计算机，我将放眼窗外，告诉你我所看见的。

我看见一个女人正在遛狗，一辆车正试图寻找一个停车位，飞机在头顶上飞过，船只在附近航行。所有这些事都是并行发生的。在本书中，我们将会学习如何将并行活动描述为相互通信的多组并行进程，并学习如何编写并发程序。

在日常用语中，并发（concurrent）、同时（simultaneous）和并行（parallel）等词几乎表示同一个意思。但在编程语言里需要做更精确的区分。具体而言，我们需要区分并发和并行程序。

如果只有一台单核的计算机，是无法在上面运行并行程序的。因为只有一个CPU，而它一次只能做一件事。然而，可以在单核计算机上运行并发程序。计算机在不同的任务之间分享时间，使人产生这些任务是并行运行的错觉。

在接下来的几节中，我们会从一些简单的并发模型起步，然后看看用并发解决问题有哪些益处，最后展示一些突出并发与并行区别的精确定义。

1.1 给并发建模

我们将从一个简单的例子入手，为一种日常情景构建并发模型。设想我看见四个人出去散步，另外还有两条狗和一大群兔子。这些人正在相互交谈，而狗则想要追逐兔子。

要在Erlang里模拟这些，需要编写四个模块，名字分别是`person`（人）、`dog`（狗）、`rabbit`（兔子）和`world`（世界）。`person`的代码会放在名为`person.erl`的文件里，看起来就像是这样：

```
-module(person).
-export([init/1]).

init(Name) -> ...
```

第1行`-module(person).`的意思是此文件包含用于`person`模块的代码。它应该与文件名一致（除了`.erl`这个文件扩展名）。模块名必须以一个小写字母开头。从技术上说，模块名是一个原子（atom）（关于原子的详细介绍，可参见3.5节）。

模块声明之后是一条导出声明。导出声明指明了模块里哪些函数可以从模块外部进行调用。它们类似于许多编程语言里的`public`声明。没有包括在导出声明里的函数是私有的，无法在模块外调用。

-export([init/1]).语法的意思是带有一个参数（/1指的就是这个意思，而不是除以1）的函数init可以在模块外调用。如果想要导出多个函数，就应该使用下面这种语法：

```
-export([FuncName1/N1, FuncName2/N2, .....]).
```

方括号[...]的意思是“列表”，因此这条声明的意思是我们想要从模块里导出一个函数列表。

我们也会给dog和rabbit编写类似的代码。

1.1.1 开始模拟

要启动程序，可以调用world:start()。它定义在一个名为world的模块里，这个模块的开头部分就像这样：

```
-module(world).
-export([start/0]).

start() ->
    Joe      = spawn(person, init, ["Joe"]),
    Susannah = spawn(person, init, ["Susannah"]),
    Dave     = spawn(person, init, ["Dave"]),
    Andy     = spawn(person, init, ["Andy"]),
    Rover    = spawn(dog,    init, ["Rover"]),
    ...
    Rabbit1  = spawn(rabbit, init, ["Flopsy"]),
   ...
```

spawn是一个Erlang基本函数，它会创建一个并发进程并返回一个进程标识符。spawn可以这样调用：

```
spawn(ModName, FuncName, [Arg1, Arg2, ..., ArgN])
```

当Erlang运行时系统执行spawn时，它会创建一个新进程（不是操作系统的进程，而是一个由Erlang系统管理的轻量级进程）。当进程创建完毕后，它便开始执行参数所指定的代码。ModName是包含想要执行代码的模块名。FuncName是模块里的函数名，而[Arg1, Arg2, ...]是一个列表，包含了想要执行的函数参数。因此，下面这个调用的意思是启动一个执行函数person:init("Joe")的进程：

```
spawn(person, init, ["Joe"])
```

spawn的返回值是一个进程标识符（PID，Process IDentifier），可以用来与新创建的进程交互。

与对象类比

Erlang里的模块类似于面向对象编程语言（OOPL，Object-Oriented Programming Language）里的类，进程则类似于OOPL里的对象（或者说类实例）。

在Erlang里，spawn通过运行某个模块里定义的函数创建一个新进程。而在Java里，new通过运行某个类中定义的方法创建一个新对象。

在OOPL里可以用一个类创建数千个类实例。类似地，在Erlang里我们可以用一个模块创建数千甚至数百万个执行模块代码的进程。所有Erlang进程都并发且独立执行，如果有一台百万核的计算机，甚至可以并行运行。

1.1.2 发送消息

启动模拟之后，我们希望在程序的不同进程之间发送消息。在Erlang里，各个进程不共享内存，只能通过发送消息来与其他进程交互。这就是现实世界里的物体行为。

假设Joe想要对Susannah说些什么。在程序里我们会编写这样一行代码：

```
Susannah ! {self(), "Hope the dogs don't chase the rabbits"}
```

`Pid ! Msg`语法的意思是发送消息`Msg`到进程`Pid`。大括号里的`self()`参数标明了发送消息的进程（在此处是`Joe`）。

1.1.3 接收消息

为了让Susannah的进程接收来自Joe的消息，要这样写：

```
receive
   {From, Message} ->
      ...
end
```

当Susannah的进程接收到一条消息时，变量`From`会绑定为`Joe`，这样Susannah就知道消息来自何处，变量`Message`则会包含此消息。

可以设想扩展一下这个模型，让狗相互发送“汪汪！兔子”的消息，而兔子则相互发送“危险！快躲起来”的消息。

这里应该记住的关键一点是：编程模型基于对现实世界的观察。之所以有三个模块（`person`、`dog`和`rabbit`）是因为例子里有三种并发的事物。`world`模块的作用是让一个顶级进程来启动这一切。创建两个狗进程是因为有两条狗，创建四个人进程是因为有四个人。程序里的消息则反映出我们在例子中观察到的消息。

我们不会扩展这个模型，而是就此止步，换个话题，来看看并发程序的一些特性。

1.2 并发的益处

并发编程可以用来提升性能，创建可扩展和容错的系统，以及编写清晰和可理解的程序来控制现实世界里的应用。下面给出了一些理由。

- 性能

  设想有两个任务：A需要10秒的执行时间，B需要15秒。在单个CPU上执行A和B需要25秒。而在拥有两个可独立运行CPU的计算机上，执行A和B只需要花费15秒。要实现这样的性能提升，必须编写并发程序。

  并行计算机一直以来都是少见而且昂贵的，但如今多核计算机已经普及开来了。顶级的处理器拥有64核，每个芯片的核的数量也有望会在短期内稳步增加。如果有合适的问题和一台64核的计算机，你的程序在它上面就可能会变快64倍，但是只有编写并发程序才能做到这一点。

  计算机行业最为紧迫的问题在于难以让传统的顺序代码在多核计算机上并行执行。Erlang不存在这个问题。20年前为顺序机器编写的Erlang程序拿到当今的多核计算机上立刻就能变得更快。

- 可扩展性

  并发程序由多个小型的独立进程组成。因此，增加进程数量和添加更多的CPU便可轻松扩展系统。在运行时，Erlang虚拟机会自动在可用的CPU之间分配进程的执行。

- 容错性

  容错性和可扩展性类似。容错的关键是独立性和硬件冗余。Erlang程序由许多小型的独立进程组成。一个进程里的错误不会导致另一个进程意外崩溃。为了防范整台计算机（或数据中心）发生故障，需要进行远程计算机的故障探测。独立进程和远程故障探测都内建在Erlang的虚拟机中。

  Erlang被设计用于构建容错式电信系统，但同样的技术也适用于构建容错式可扩展的Web系统和云服务。

- 清晰性

  在现实世界里，万事是并行发生的，但在大多数编程语言里事情是顺序发生的。现实世界并行性和编程语言顺序性之间的不匹配使得用顺序性语言编写控制现实问题的程序变得困难。在Erlang里可以直观地将现实世界的并行性映射到Erlang的并发性上。这就使代码变得清晰和易于理解。

  了解这些益处之后，我们接下来会试着更准确地解释并发和并行这两个概念，为在后面的章节里讨论这些术语打下基础。

1.3 并发程序和并行计算机

现在我要卖弄一下学者风范，尝试给出并发和并行这类术语的准确含义。先来区分一下并发程序（也就是如果有并行计算机就可能会跑得更快的程序）和多核（或CPU）的并行计算机。

- 并发程序是一种用并发编程语言编写的程序。编写并发程序是为了提升性能、可扩展性和容错性。

- 并发编程语言拥有专门用于编写并发程序的语言结构。这些结构是编程语言的主要部分，在所有操作系统上都有着相同的表现。
- 并行计算机是一种有多个处理单元（CPU或核心）同时运行的计算机。

Erlang里的并发程序是由互相通信的多组顺序进程组成的。一个Erlang进程就是一个小小的虚拟机，可以执行单个Erlang函数。别把它和操作系统的进程相混淆。

要用Erlang编写一个并发程序，必须确定一组用来解决问题的进程。这种确定进程的做法被称为并发建模。它类似于在编写面向对象程序时确定所需对象的技艺。

在面向对象设计里，选出解决问题所需的对象是一个公认的难题。并发建模也是如此。选出正确的进程可能会很困难。好的进程模型能实现设计，差的模型则会毁了设计。

编写完并发程序之后，可以在并行计算机上运行。我们可以在单台多核计算机上运行，还可以在一组联网的计算机上运行，也可以在云端运行。

我们的并发程序是否真的会在并行计算机上并行运行？有时候你很难知道。在多核计算机上，操作系统也许会决定关闭一个核心来节能。在云端，某个计算也许会被挂起并转移到一台新机器上。这些情况不在我们的控制范围内。

现在我们了解了并发程序和并行计算机之间的区别。并发性与软件结构有关，而并行性与硬件有关。下面来看看顺序和并发编程语言之间的区别。

1.4　顺序和并发编程语言

编程语言有两种：顺序和并发。顺序语言被设计用于编写顺序程序，没有描述并发计算的语言结构。并发编程语言被设计用于编写并发程序，语言本身带有表达并发性的特殊结构。

在Erlang里，并发性由Erlang虚拟机提供，而非操作系统或任何的外部库。在大多数顺序编程语言里，并发性都是以接口的形式提供，指向主机操作系统的内部并发函数。

区分基于操作系统的并发和基于语言的并发很重要，因为如果使用基于操作系统的并发，那么程序在不同的操作系统上就会有不同的工作方式。Erlang的并发在所有操作系统上都有着相同的工作方式。要用Erlang编写并发程序，只需掌握Erlang，而不必掌握操作系统的并发机制。

在Erlang里，进程和并发是我们可以用来定型和解决问题的工具。这让细粒度控制程序的并发结构成为可能，而用操作系统的进程是很难做到的。

1.5　小结

本章介绍了本书的中心主题，讨论了用并发的方法编写高性能、可扩展和容错的软件，但并未涉及如何实现这一切的细节。在下一章，我们会快速游历Erlang，并编写第一个并发程序。

第 2 章

Erlang速览

在这一章里，我们会创建第一个并发程序。制作一个文件服务器，使其拥有两个并发进程：一个进程代表服务器，另一个代表客户端。

我们将从一个小的Erlang子集着手，这样就能展示一些总体的原则，而不会在细节中挣扎。但至少要理解如何在shell里运行代码和编译模块。作为起步了解这些就够了。

学习Erlang的最佳方式是把示例输入运行中的Erlang系统，看看你是否能复制书里的内容。要安装Erlang，请访问http://joearms.github.com/installing.html。尽量让安装指南保持最新的状态，这有点困难，因为有许多不同的平台，它们的配置也多种多样。如果指南出错或者过时了，请发送一封邮件到Erlang的邮件列表，我们会尽力提供帮助。

2.1 Shell

你的大多数时间会花费在Erlang的shell[①]里。输入一条表达式，shell就会执行这条表达式并显示出结果。

```
$ erl
Erlang R16B ...
Eshell V5.9  (abort with ^G)
1> 123456 * 223344.
27573156864
```

那么，上面发生了什么？$是操作系统提示符。输入的命令erl启动了Erlang shell。Erlang shell以横幅信息和计数提示符1>作为响应。然后输入一个表达式，并得到了执行和显示。请注意每一条表达式都必须以一个句号后接一个空白字符结尾。在这个上下文环境里，空白是指空格、制表（Tab）或者回车符。

初学者经常会忘记用句号加空白来结束表达式。可以把命令想象成英语句子。英语句子通常以句号结尾，所以很容易记住。

① 类似于Windows操作系统的命令行界面。——译者注

2.1.1　=操作符

可以用=操作符给变量赋值（严格来说是给变量*绑定*一个值），就像这样：

```
2> X = 123.
123
3> X * 2.
246
```

如果试图改变变量的值，奇怪的事情就会发生。

```
4> X = 999.
** exception error: no match of right hand side value 999
```

这是第一件让人惊奇的事。不能*重新绑定*变量。Erlang是一种函数式语言，所以一旦定义了`X = 123`，那么X*永远*就是123，不允许改变！

不必担心，这其实是一个优点，而不是问题。相比那些同一个变量可以在程序生命周期里获得多个不同值的程序，变量一旦设置就不能改动的程序要容易理解得多。

当看到一个表达式像`X = 123`时，它的意思看似“将整数123赋予变量X”，但这种解读是不正确的。=不是一个赋值操作符，它实际上是一个*模式匹配操作符*。详情请参见3.3.2节。

与其他函数式编程语言一样，Erlang的变量只能绑定一次。绑定变量的意思是给变量一个值，一旦这个值被绑定，以后就不能改动了。

如果你习惯了命令式语言，可能会对这个概念感到不可思议。在命令式语言里，变量其实是伪装起来的内存地址。某个程序里的X其实就是内存某处的数据项地址。定义X=12时，改变的是地址X处的内存值，但在Erlang里，变量X代表的是一个永远不能改变的值。

2.1.2　变量和原子的语法

请注意Erlang的变量以大写字母开头。所以X、`This`和`A_long_name`都是变量。以小写字母开头的名称（比如`monday`或`friday`）不是变量，而是符号常量，它们被称为*原子*（atom）。

如果你在任何时候看到或写出像`x = 123`这样的表达式（注：这里的x是小写的，如果没有注意到的话），那么几乎可以肯定这是一个错误。如果在shell里这么做，会立即得到应答。

```
1> abc=123.
** exception error: no match of right hand side value 123
```

但如果这样一行表达式深深隐藏在代码里面，就可能会让程序崩溃，所以要小心。大多数编辑器（比如Emacs和Eclipse编辑器）会用不同的颜色显示代码里的原子和变量，所以区分它们很容易。

在阅读下一节之前，请试着启动shell并输入一些简单的算术表达式。在当前阶段，如果哪里出了问题，只需要输入`Control+C`后接`a`（代表“abort”，即中止）来退出shell，然后从操作系统提示符处重启shell即可。

我们了解了如何启动和停止shell，并用它来执行简单的表达式。还看到了函数式编程语言与

命令式编程语言的根本区别。在函数式语言里，变量不能改变，在命令式语言里却可以。

2.2　进程、模块和编译

Erlang程序是由许多并行的进程构成的。进程负责执行模块里定义的函数。模块则是扩展名为`.erl`的文件，运行前必须先编译它们。编译某个模块之后，就可以在shell或者直接从操作系统环境的命令行里执行该模块中的函数了。

下面几节将介绍如何在shell或操作系统命令行里编译模块和执行函数。

2.2.1　在 shell 里编译并运行 Hello World

请制作一个带有以下内容的`hello.erl`文件：

hello.erl

```
-module(hello).
-export([start/0]).

start() ->
    io:format("Hello world~n").
```

为了编译并运行它，我们从保存`hello.erl`的目录里启动Erlang shell，然后执行下面的操作：

```
$ erl
Erlang R16B ...
1> c(hello).
{ok,hello}
2> hello:start().
Hello world
ok
3> halt().
$
```

`c(hello)`命令编译了`hello.erl`文件里的代码。`{ok, hello}`的意思是编译成功。现在代码已准备好运行了。第2行里执行了`hello:start()`函数。第3行里停止了Erlang shell。

在shell里进行操作的优点是只要平台为Erlang所支持，这种编译和运行程序的方法就一定可用。在操作系统的命令行里的操作可能会因平台的不同而有所差别。

2.2.2　在 Erlang shell 外编译

也可以在操作系统的命令行里编译和运行前一个例子中的代码，就像下面这样：

```
$ erlc hello.erl
$ erl -noshell -s hello start -s init stop
Hello world
```

`erlc`从命令行启动了Erlang编译器。编译器编译了`hello.erl`里的代码并生成一个名为

`hello.beam`的目标代码文件。

`$erl -noshell ...`命令加载了`hello`模块并执行`hello:start()`函数。随后，它执行了`init:stop()`，这个表达式终止了Erlang会话。

在Erlang shell之外运行Erlang编译器（`erlc`）是编译Erlang代码的首选方式。可以在Erlang shell里编译模块，但要这么做必须首先启动Erlang shell。使用`erlc`的优点在于自动化。我们可以在rakefile或makefile内运行`erlc`来自动化构建过程。

刚开始学习Erlang时，建议你用Erlang shell进行所有的操作，这样就能熟悉编译和运行代码的细节。高级用户会更喜欢自动编译，对Erlang shell的使用也会变少。

2.3　你好，并发

我们已经了解了如何编译单个模块。那么如何编写并发程序呢？Erlang的基本并发单元是进程（process）。一个进程是一个轻量级的虚拟机，只能通过发送和接收消息来与其他进程通信。如果你想让一个进程做点什么，就要给它发送一个消息，还可能需要等待答复。

将要编写的第一个并发程序是一个文件服务器。要在两台机器之间传输文件，需要两个程序：第一台机器上运行的客户端和第二台机器上运行的服务器。为了实现这一点，我们将制作两个模块：`afile_client`和`afile_server`。

2.3.1　文件服务器进程

文件服务器由一个名为`afile_server`的模块实现。这里再提醒一下，进程和模块类似于对象和类。用于进程的代码包含在模块里，要创建一个进程，需要调用`spawn(...)`，创建进程的实际操作由这个基本函数完成。

afile_server.erl

```
-module(afile_server).
-export([start/1, loop/1]).

start(Dir) -> spawn(afile_server, loop, [Dir]).

loop(Dir) ->
    receive
        {Client, list_dir} ->
            Client ! {self(), file:list_dir(Dir)};
        {Client, {get_file, File}} ->
            Full = filename:join(Dir, File),
            Client ! {self(), file:read_file(Full)}
    end,
    loop(Dir).
```

这段代码的结构非常简单。如果略去大部分细节，它看起来就会像这样：

```
loop(Dir) ->
   %% 等待指令
   receive
      Command ->
        … 做点什么 …
   end,
   loop(Dir).
```

这就是用Erlang编写无限循环的方法。变量`Dir`包含了文件服务器当前的工作目录。我们在这个循环内等待指令，接收到指令时我们会遵从，然后再次调用自身来获取下一个指令。

答疑解惑　不用担心最后的自身调用，这不会耗尽栈空间。Erlang对代码采用了一种所谓“尾部调用”的优化，意思是此函数的运行空间是固定的。这是用Erlang编写循环的标准方式，只要在最后调用自身即可。

另一点要注意的是，`loop`函数永远不会返回。在顺序编程语言里，必须要极其小心避免无限循环，因为只有一条控制线，如果这条线卡在循环里就有麻烦了。Erlang则没有这个问题。服务器只是一个在无限循环里处理请求的程序，与我们想要执行的其他任务并行运行。

现在仔细查看接收语句。回忆一下，它看起来就像这样：

afile_server.erl

```
receive
    {Client, list_dir} ->
        Client ! {self(), file:list_dir(Dir)};
    {Client, {get_file, File}} ->
        Full = filename:join(Dir, File),
        Client ! {self(), file:read_file(Full)}
end,
```

这段代码的意思是如果接收到`{Client, list_dir}`消息，就应该回复一个文件列表；如果接收到的消息是`{Client, {get_file, File}}`，则回复这个文件。作为模式匹配过程的一部分，`Client`变量在收到消息时会被绑定。

这段代码非常紧凑，所以很容易忽略所发生的细节。这段代码里有三个要点需要加以注意。

- 回复给谁

  所有接收的消息都包含变量`Client`，它是发送请求进程的进程标识符，也是应该回复的对象。如果想要得到一条消息的回复，最好说明一下回复应该发给谁。就像在信件里写明姓名和地址，如果不说明信件来自何处，就永远得不到回复。

- `self()`的用法

  服务器发送的回复包含了参数`self()`（在这个案例里`self()`是服务器的进程标识符）。这个标识符被附在消息中，使客户端可以检查收到的消息的确来自服务器，而不是其他某个进程。

- 模式匹配被用于选择消息

  接收语句内部有两个模式。可以这样编写：

```
receive
    Pattern1 ->
      Actions1;
    Pattern2 ->
      Actions2 ->
    ...
end
```

Erlang编译器和运行时系统会正确推断出如何在收到消息时运行适当的代码。不需要编写任何的`if-then-else`或`switch`语句来设定该做什么。这是模式匹配带来的乐趣之一，会为你节省大量工作。

可以像下面这样在shell里编译和测试这段代码：

```
1> c(afile_server).
{ok,afile_server}
2> FileServer = afile_server:start(".").
<0.47.0>
3> FileServer ! {self(), list_dir}.
{<0.31.0>,list_dir}
4> receive X -> X end.
{<0.47.0>,
 {ok,["afile_server.beam","processes.erl","attrs.erl","lib_find.erl",
     "dist_demo.erl","data1.dat","scavenge_urls.erl","test1.erl",
     ...]}}
```

来看看相关细节。

```
1> c(afile_server).
{ok,afile_server}
```

编译`afile_server.erl`文件所包含的`afile_server`模块。编译很成功，所以“编译”函数c的返回值是`{ok, afile_server}`。

```
2> FileServer = afile_server:start(".").
<0.47.0>
```

`afile_server:start(Dir)`调用`spawn(afile_server, loop, [Dir])`。这就创建出一个新的并行进程来执行函数`afile_server:loop(Dir)`并返回一个进程标识符，可以用它来与此进程通信。

`<0.47.0>`是文件服务器进程的进程标识符。它的显示方式是尖括号内由句号分隔的三个整数。

注意　每次运行这个程序时，进程标识符都会改变。因此，`<0.47.0>`里的数字在不同的会话里将会是不同的。

```
3> FileServer ! {self(), list_dir}.
{<0.31.0>,list_dir}
```

这里给文件服务器进程发送了一条`{self(), list_dir}`消息。`Pid ! Message`的返回值被规定为`Message`，因此shell打印出`{self(),list_dir}`的值，即`{<0.31.0>, list_dir}`。

<0.31.0>是Erlang shell自身的进程标识符，它被包括在消息内，告知文件服务器应该回复给谁。

```
4> receive X -> X end.
{<0.47.0>,
{ok,["afile_server.beam","processes.erl","attrs.erl","lib_find.erl",
     "dist_demo.erl","data1.dat","scavenge_urls.erl","test1.erl",
     ...]}
```

receive X -> X end接收文件服务器发送的回复。它返回元组{<0.47.0>, {ok, ...}。该元组的第一个元素<0.47.0>是文件服务器的进程标识符。第二个参数是file:list_dir(Dir)函数的返回值，它在文件服务器进程的接收循环里得出。

2.3.2 客户端代码

文件服务器通过一个名为afile_client的客户端模块进行访问。这个模块的主要目的是为了隐藏底层通信协议的细节。客户端代码的用户可以通过调用此客户端模块导出的ls和get_file函数来传输文件。这就能够自由改变底层的协议而不会影响到客户端代码API部分。

afile_client.erl

```
-module(afile_client).
-export([ls/1, get_file/2]).

ls(Server) ->
    Server ! {self(), list_dir},
    receive
        {Server, FileList} ->
            FileList
    end.

get_file(Server, File) ->
    Server ! {self(), {get_file, File}},
    receive
        {Server, Content} ->
            Content
    end.
```

如果对比afile_clien与afile_server的代码，就会发现一种美妙的对称性。只要客户端里有Server ! ...这类send语句，服务器里就会有receive模式，反之亦然。

```
receive
    {Client, Pattern} ->
         ...
end
```

现在要重启shell并重新编译所有代码，展示客户端和服务器如何共同工作。

```
1> c(afile_server).
{ok,afile_server}
```

```
2> c(afile_client).
{ok,afile_client}
3> FileServer = afile_server:start(".").
<0.43.0>
4> afile_client:get_file(FileServer,"missing").
{error,enoent}
5> afile_client:get_file(FileServer,"afile_server.erl").
{ok,<<"-module(afile_server).\n-export([start/1])....}
```

现在在shell里运行的代码和之前代码的唯一区别，是把接口程序抽象出来放入单独的模块里。我们隐藏了客户端和服务器之间消息传递的细节，因为没有其他程序对此感兴趣。

到目前为止，你看到了一个完整文件服务器的基础部分，但它尚未完成。还有很多的细节问题，涉及启动和停止服务器、连接某个套接字（socket）等，在这里不会提及。

从Erlang的角度看，如何启动和停止服务器，连接套接字，从错误中恢复等都是琐碎的细节。问题的本质在于创建并行进程，以及发送和接收消息。

在Erlang里用进程来构建问题的解决方案。思考进程的结构（也就是说哪些进程间相互有联系），思考进程间传递的消息以及消息包含何种信息是思考和编程方式的中心。

2.3.3　改进文件服务器

我们开发的文件服务器包括了运行在同一台机器上的两个相互通信的进程，展示了编写并发程序所需的一些基本要素。对真正的服务器而言，客户端和服务器会运行在不同的机器上，所以必须设法让进程间的消息不仅能在同一个Erlang节点的进程之间传递，也能在物理上隔开的机器上的Erlang进程之间进行。

第17章会介绍如何将TCP传输层用于进程通信，而14.4.2节将介绍如何直接用分布式Erlang实现文件服务器。

在这一章里，我们了解了如何在shell里执行一些简单的操作，编译一个模块，以及用`spawn`、`send`和`receive`这三个基本函数创建一个简单的双进程并发程序。

这就是本书的第一部分。第二部分会更详细地讨论顺序编程，直到第12章再回到并发编程上来。在下一章，我们将开始学习顺序编程，详细了解shell、模式匹配和Erlang的基本数据类型。

2.4　练习

现在也许是个好机会来检查一下你对目前所做的有几分了解。

(1) 启动并停止Erlang shell。

(2) 在Erlang shell里输入一些命令。不要忘了以句号和空白结束命令。

(3) 对`hello.erl`做一些小小的改动。在shell里编译并运行它们。如果有错，中止Erlang shell并重启shell。

(4) 运行文件客户端和服务器代码。加入一个名为`put_file`的命令。你需要添加何种消息？学习如何查阅手册页。查阅手册页里的`file`模块。

Part 2 第二部分

顺序编程

本部分涉及如何编写顺序的 Erlang 程序。我们将介绍所有的顺序 Erlang 知识，讨论编译和运行程序的各种方法，以及用类型系统来描述 Erlang 函数的类型和静态检测编程错误。

第 3 章

基本概念

这一章是Erlang编程的基础。无论是并行还是顺序，所有的Erlang程序都会用到模式匹配、一次性变量赋值和Erlang用于表示数据的基本类型。

在这一章里，我们会用Erlang shell对系统做一些尝试，看看它有什么样的行为。首先介绍shell。

3.1　启动和停止 Erlang shell

在Unix系统（包括Mac OS X）里，需要从命令提示里启动Erlang shell；在Windows系统里则是点击Window开始菜单里的Erlang图标。

```
$ erl
Erlang R16B (erts-5.10.1) [source] [64-bit] [smp:4:4] [async-threads:10]
    [hipe] [kernel-poll:false]
Eshell V5.10.1  (abort with ^G)
1>
```

这是启动Erlang shell的Unix命令。shell以横幅信息作为响应，告诉你正在运行哪个版本的Erlang。停止系统最简单的方式是按Ctrl+C（Windows里是Ctrl+Break[①]）后接a（“abort”的简写），就像下面这样：

```
BREAK: (a)bort (c)ontinue (p)roc info (i)nfo (l)oaded
       (v)ersion (k)ill (D)b-tables (d)istribution
a
$
```

输入a会立即停止系统，这可能导致某些数据的损坏。要实现受控关闭，可以输入q()（“quit”的简写）。

```
1> q().
ok
$
```

① 即键盘上的“Pause|Break”键。——译者注

这样就以一种受控的方式停止了系统。所有打开的文件都被刷入缓存并关闭，数据库（如果正在运行的话）会被停止，所有的应用程序都以有序的方式关停。`q()`是`init:stop()`命令在shell里的别名。

要立即停止系统，应执行表达式`erlang:halt()`。

如果这些方法都不管用，请阅读10.4.1节的相关内容。

3.1.1 在shell里执行命令

当shell准备好接受表达式时，会打印出命令提示。

```
1> X = 20.
20
```

这次对话从命令1开始（也就是shell打印出的`1>`）。它的意思是启动了一个新的Erlang shell。每当在本书里看到以`1>`开头的对话时，就必须启动一个新的shell才能准确再现本书里的示例。当某个示例以大于1的提示数字开头时，就暗示此shell会话是之前示例的延续，此时无需启动新shell。

在提示处输入一个表达式。shell执行了这个表达式并打印出结果。

```
2> X + 20. % and this is a comment
40
```

shell又打印出了一个提示，这一次是用于表达式2（因为每次输入一个新表达式，命令数字就会增加）。

在第2行里，百分号字符（%）代表一段注释的起点。从百分号到行尾的所有文字都被当作注释，shell和Erlang编译器会忽略它们。

现在也许是拿shell来做实验的好机会。请按照书中文字的显示方式准确输入示例里的这些表达式，然后检查是否得到了与本书相同的结果。有些命令序列可以多次输入，而其他的只能输入一次，因为它们依赖于之前的命令。如果任何地方出了错，最好的方法是中止shell，然后新开一个shell重试。

3.1.2 可能出错的地方

你不能把在本书里读到的一切都输进shell里。Erlang模块里的语法形式不是表达式，不能被shell理解。具体来说，不能在shell里输入附注，附注是以连字符开头的事物（例如`-module`和`-export`）。

另一个可能出错的地方是，已开始输入一些引号内文字（也就是以单引号或双引号开头）但尚未输入与开始引号相匹配的结束引号。

如果这些错误发生了，最好的做法是输入一个额外的结束引号，后接句号和空白来完成命令。

高级技巧：可以启动和停止多个shell。详情请参见10.4.3节。

3.1.3　在Erlang shell里编辑命令

Erlang shell包含一个内建的行编辑器。它能部分理解流行的Emacs编辑器所使用的行编辑命令。只需按几次键就能重新调用和编辑之前的行。下面展示了可用的命令（注意^Key的意思是应该按下Ctrl+Key）：

命　令	说　明
^A	行首
^D	删除当前字符
^E	行尾
^F或右箭头键	向前的字符
^B或左箭头键	向后的字符
^P或上箭头键	前一行
^N或下箭头键	下一行
^T	调换最近两个字符的位置
Tab	尝试扩展当前模块或函数的名称

使用经验越来越丰富后，你会明白shell真的是一个很强大的工具。最棒的是，当开始编写分布式程序时，你会发现可以挂接一个shell到集群里另一个Erlang节点上运行的Erlang系统，甚至还可以生成一个安全shell（secure shell，即ssh）直接连接远程计算机上运行的Erlang系统。通过它，可以与Erlang节点系统中任何节点上的任何程序进行交互。

3.2　简单的整数运算

先来计算一些算术表达式的值。

```
1> 2 + 3 * 4.
14
2> (2 + 3) * 4.
20
```

你会看到Erlang遵循算术表达式的一般规则，因此2 + 3 * 4的意思是2 + (3 * 4)，而不是(2 + 3) * 4。

Erlang可以用任意长度的整数执行整数运算。在Erlang里，整数运算是精确的，因此无需担心运算溢出或无法用特定字长（word size）来表示某个整数。

为什么不试试呢？你可以计算一些非常大的数字来向朋友炫耀一下。

```
3> 123456789 * 987654321 * 112233445566778899 * 998877665544332211.
13669560260321809985966198898925761696613427909935341
```

可以用多种方式输入整数（详情参见8.19.1节）。下面这个表达式使用了十六进制和三十二进制的记数法：

```
4> 16#cafe * 32#sugar.
1577682511434
```

3.3　变量

可以把某个命令的结果保存在变量里。

```
1> X = 123456789.
123456789
```

在第1行里给变量X指派一个值，shell在下一行里打印出了这个变量的值。

请注意所有变量名都必须以大写字母开头。

如果想知道一个变量的值，只需要输入变量名。

```
2> X.
123456789
```

X既然已经有一个值了，就可以使用它。

```
3> X*X*X*X.
232305722798259244150093798251441
```

但是，如果试图给变量X指派一个不同的值，就会得到一条错误消息。

```
4> X = 1234.
** exception error: no match of right hand side value 1234
```

一次性赋值就像是代数

记得上学的时候，数学老师曾说过："如果一个等式中有好几处*X*，那么所有*X*都是一样的。"我们就是这么解方程的：如果知道$X+Y=10$且$X-Y=2$，那么这两个等式里的*X*就都等于6，*Y*都等于4。

但当我学习我的第一门编程语言时，看到的却是这个：

```
X = X + 1
```

所有人都在抗议，说："这是不可能的！"但是老师说是我们错了，我们必须忘记数学课里学到的那些。

在Erlang里，变量就像数学里的那样。当关联一个值与一个变量时，所下的是一种断言，也就是事实陈述。这个变量具有那个值，仅此而已。

为了解释这里发生了什么，我不得不破坏对`X = 1234`这条简单语句所带有的两种假定。

- 首先，X不是一个变量，不是你习惯的Java和C等语言里的概念。
- 其次，=不是一个赋值操作符，而是一个模式匹配操作符。

这多半是你初学Erlang时遇到的最难懂的部分之一，所以我们来深入探讨一下。

3.3.1 Erlang的变量不会变

Erlang的变量是一次性赋值变量（single-assignment variable）。顾名思义，它们只能被赋值一次。如果试图在变量被设置后改变它的值，就会得到一个错误（事实上，你将得到的是我们刚才看到的“badmatch”，即不匹配错误）。已被指派一个值的变量称为绑定变量，否则称为未绑定变量。

当Erlang看到像`X = 1234`这样的一条语句且X之前未被绑定时，它就会给变量X绑定1234这个值。X在绑定前可以接受任何值，它只是一个等待填充的空槽。但是，一旦得到一个值，就会永远保留它。

此时此刻，你可能很想知道为什么用变量这个名称。这里有两个原因。

- 它们是变量，但它们的值只能被改变一次（也就是说，它们从未绑定改变为具有一个值）。
- 它们看上去像是传统编程语言里的变量，因此当看到像这样开头的一行代码时：

  ```
  X = ...  %% '...' 意指'没有显示的代码'
  ```

  就会想：“啊哈，我知道这是什么。X是一个变量，=是一个赋值操作符。”所以说X应该是一个变量，而=是一个赋值操作符。

事实上，=是一个模式匹配操作符，它在X为未绑定变量时的表现类似于赋值。

最后，变量的作用域是它定义时所处的语汇单元。因此，如果X被用在一条单独的函数子句之内，它的值就不会“逃出”这个子句。没有同一函数的不同子句共享全局或私有变量这种说法。如果X出现在许多不同的函数里，那么所有这些X的值都是不相干的。

3.3.2 变量绑定和模式匹配

在Erlang里，变量获得值是一次成功模式匹配操作的结果。

在大多数语言里，=表示一个赋值语句。而在Erlang里，=是一次模式匹配操作。`Lhs = Rhs`的真正意思是：计算右侧（`Rhs`）的值，然后将结果与左侧（`Lhs`）的模式相匹配。

变量（比如X）是模式的一种简单形式。如之前所说的，变量只能赋值一次。我们第一次说`X = SomeExpression`时，Erlang对自己说：“我要做些什么才能让这条语句为真？”因为X还没有值，它可以绑定X到`SomeExpression`这个值上，这条语句就成立了。

如果后期我们说`X = AnotherExpression`，那么只有在`SomeExpression`和`AnotherExpression`相等的情况下匹配才会成功。这里有一些例子：

```
1> X = (2+4).
6
```

X在这条语句之前没有值，因此模式匹配成功，X被绑定为6。

```
2> Y = 10.
10
```

类似地，Y被绑定为10。

```
3> X = 6.
6
```

这与第1行略有不同。在计算这个表达式之前，X等于6，因此匹配成功，shell则打印出了这个表达式的值，也就是6.

```
4> X = Y.
** exception error: no match of right hand side value 10
```

在计算这个表达式之前，X等于6，Y等于10。6与10不相等，因此会打印出一条错误消息。


```
5> Y = 10.
10
```

这次模式匹配成功，因为Y等于10。

```
6> Y = 4.
** exception error: no match of right hand side value 4
```

这次失败，因为Y等于10。

你可能会觉得我在这一点上过于啰嗦了。现在这个阶段，=左边的所有模式都仅仅是绑定或未绑定的变量，但在后面将会看到，我们可以制作任意复杂的模式来用=操作符进行匹配。我会在介绍元组和列表（它们被用于保存复合数据项）之后再回到这个主题上。

没有副作用意味着可以让程序并行

可以修改的内存区域有个专业术语叫作可变状态（mutable state）。Erlang是一种函数式编程语言，具有不可变状态。

本书后面将介绍如何为多核CPU编程，看看不可变状态所带来的显著效果。

如果使用C或Java这些传统编程语言来为多核CPU编程，就不得不对付共享内存的问题。为了不破坏共享内存，访问时必须给这些内存加锁。访问共享内存的程序在操作共享内存时万万不可崩溃。

Erlang里没有可变状态，没有共享内存，也没有锁。这让程序并行变得简单了。

3.3.3 为什么一次性赋值让程序变得更好

在Erlang里，变量只不过是对某个值的引用：Erlang的实现方式用指针代表绑定变量，指向一个包含值的存储区。这个值不能被修改。

不能修改变量这一事实极其重要，和C或Java这些命令式语言里的变量行为存在区别。

使用不可变变量简化了调试工作。要理解为什么是这样，我们必须问自己错误是什么，以及错误是如何表现出来的。

程序出错的一个常见发现方式是我们看到某个变量有着意料之外的值。一旦知道是哪个变量出了错，就必须检查程序，找到绑定变量的地方。因为Erlang的变量是不可变的，所以生成此变

量的代码必然是错误的。而在命令式语言里，变量可以被多次修改，因此每一个修改变量的地方都有可能是错误发生之处。在Erlang里则只需要查看一处。

此时此刻，你可能想知道：没有可变变量将如何编程？在Erlang里怎样表达X = X + 1这类概念？Erlang的方式是创建一个名字未被使用过的新变量（比方说X1），然后编写X1 = X + 1。

3.4　浮点数

让我们试着用浮点数做一些计算。

```
1> 5/3.
1.6666666666666667
```

第1行的行尾数字是整数3。句号表示表达式的结束，而不是小数点。如果我想写的是浮点数，我会写作3.0。

当你用/给两个整数做除法时，结果会自动转换成浮点数。因此，5/3的值是1.6666666666666667。

```
2> 4/2.
2.0
```

尽管4能被2整除，但是结果仍然是一个浮点数，而不是整数。要从除法里获得整数结果，我们必须使用操作符div和rem。

```
3> 5 div 3.
1
4> 5 rem 3.
2
5> 4 div 2.
2
```

N div M是让N除以M然后舍去余数。N rem M是N除以M后剩下的余数。

Erlang在内部使用64位的IEEE 754-1985浮点数，因此使用浮点数的程序会存在和C等语言一样的浮点数取整与精度问题。

3.5　原子

在Erlang里，原子被用于表示常量值。

如果你熟悉C或Java里的枚举类型，或者Scheme或Ruby里的符号，说明你已经用过与原子非常相似的数据类型了。

C程序员会很熟悉一个惯例：用符号常量来让程序能自我描述。一个典型的C程序会在包含文件里定义一组全局常量（这个文件就是由众多常量定义组成的）。举个例子，一个名为glob.h的文件可能会包含这些：

```
#define OP_READ 1
#define OP_WRITE 2
#define OP_SEEK 3
...
#define RET_SUCCESS 223
...
```

使用这些符号常量的典型C代码可能是这样的：

```
#include "glob.h"
int ret;
ret = file_operation(OP_READ, buff);
if( ret == RET_SUCCESS ) { ... }
```

在C程序里，这些常量的值无关紧要，而在这里它们之所以值得注意仅仅是因为其各不相同，可以用来比较是否相等。这个程序对应的Erlang版本可以是这样：

```
Ret = file_operation(op_read, Buff),
if
    Ret == ret_success ->
       ...
```

在Erlang里，原子是全局性的，而且不需要宏定义或包含文件就能实现。

假设想要编写一个对星期几进行操作的程序。要实现它，我们会用`monday`（星期一）和`tuesday`（星期二）等原子来代表它们。

原子以小写字母开头，后接一串字母、数字、下划线（_）或at（@）符号，例如`red`、`december`、`cat`、`meters`、`yards`、`joe@somehost`和`a_long_name`。

原子还可以放在单引号（'）内。可以用这种引号形式创建以大写字母开头（否则会被解释成变量）或包含字母数字以外字符的原子，例如`'Monday'`、`'Tuesday'`、`'+'`、`'*'`和`'an atom with spaces'`。甚至可以给无需引号的原子加上引号，因此`'a'`和`a`的意思完全一致。在某些语言里，单引号和双引号可以互换使用。Erlang里不是这样。单引号的用法如前面所示，双引号用于给字符串字面量（string literal）定界。

一个原子的值就是它本身。所以，如果输入一个原子作为命令，Erlang shell就会打印出这个原子的值。

```
1> hello.
hello
```

你可能会觉得讨论原子或整数的值有点奇怪。但因为Erlang是一种函数式编程语言，每个表达式都必须有一个值。这包括了整数和原子，它们只不过是极其简单的表达式。

3.6 元组

如果想把一些数量固定的项目归组成单一的实体，就会使用*元组*（tuple）。创建元组的方法是用大括号把想要表示的值括起来，并用逗号分隔它们。举个例子，如果想要表示某人的名字和

身高，就可以用{joe, 1.82}。这个元组包含了一个原子和一个浮点数。

元组类似C里面的结构（struct），区别在于元组是匿名的。在C里，类型为point的变量P可以这样声明：

```
struct point {
  int x;
  int y;
} P;
```

我们使用点操作符来访问C结构里的各个字段。因此，要设置这个坐标点的x和y值，可以这么写：

```
P.x = 10; P.y = 45;
```

Erlang没有类型声明，因此要创建一个“坐标点”，只需要这么写：

```
P = {10, 45}
```

这样就创建了一个元组并把它绑定到变量P上。与C结构不同，元组里的字段没有名字。因为这个元组只包含一对整数，所以必须记住它的用途是什么。为了更容易记住元组的用途，一种常用的做法是将原子作为元组的第一个元素，用它来表示元组是什么。因此，我们会写成{point, 10, 45}而不是{10, 45}，这就使程序的可理解性大大增加了。这种给元组贴标签的方式不是语言所要求的，而是一种推荐的编程风格。

元组还可以嵌套。假如想要表示某人的一些情况（名字、身高、鞋码和眼睛颜色），就可以像下面这么写：

```
1> Person = {person, {name, joe}, {height, 1.82},
    {footsize, 42}, {eyecolour, brown}}.
{person, {name, joe}, {height, 1.82}, {footsize, 42}, {eyecolour, brown}}
```

请注意我们是如何将原子同时用于标明字段和（在name和eyecolour里）作为字段值的。

3.6.1　创建元组

元组会在声明它们时自动创建，不再使用时则被销毁。

Erlang使用一个垃圾收集器来回收所有未使用的内存，这样就不必担心内存分配的问题了。

如果在构建新元组时用到变量，那么新的元组会共享该变量所引用数据结构的值。下面是一个例子：

```
2> F = {firstName, joe}.
{firstName,joe}
3> L = {lastName, armstrong}.
{lastName,armstrong}
4> P = {person, F, L}.
{person,{firstName,joe},{lastName,armstrong}}
```

如果试图用未定义的变量创建数据结构，就会得到一个错误。

```
5> {true, Q, 23, Costs}.
** 1: variable 'Q' is unbound **
```

这句的意思就是变量Q未定义。

3.6.2　提取元组的值

之前说过，虽然=看上去像是赋值语句，但其实不是，它是一个模式匹配操作符。你可能好奇我们为什么这么讲究。原因在于，模式匹配是Erlang的根基，在众多不同的任务中都会用到。它被用于从数据结构里提取值，控制函数内部的流程，在并行程序里给进程发消息时，还会用它选择该处理哪些消息。

如果想从某个元组里提取一些值，就会使用模式匹配操作符=。

再回头看看表示坐标点的那个元组。

```
1> Point = {point, 10, 45}.
{point, 10, 45}.
```

假如想把Point里的字段提取到变量X和Y里，可以像下面这样写：

```
2> {point, X, Y} = Point.
{point,10,45}
3> X.
10
4> Y.
45
```

在命令2里，X绑定了10，Y绑定了45。根据规定，表达式Lhs = Rhs的值是Rhs，因此shell打印出{point,10,45}。

如你所见，等号两侧的元组必须有相同数量的元素，而且两侧的对应元素必须绑定为相同的值。

现在假设输入了这样的语句：

```
5> {point, C, C} = Point.
** exception error: no match of right hand side value {point,10,45}
```

模式{point, C, C}与{point, 10, 45}不匹配，因为C不能同时是10和45。因此，这次模式匹配失败，系统打印出了错误消息。

下面这个例子里，模式{point, C, C}则是匹配的：

```
6> Point1 = {point,25,25}.
{point,25,25}
7> {point, C, C} = Point1.
{point,25,25}
8> C.
25
```

如果有一个复杂的元组，就可以编写一个与该元组形状（结构）相同的模式，并在待提取值的位置加入未绑定变量来提取该元组的值。

为了说明这一点，首先定义一个包含复杂数据结构的变量Person。

```
1> Person={person,{name,joe,armstrong},{footsize,42}}.
{person,{name,joe,armstrong},{footsize,42}}
```

现在将编写一个模式来提取此人姓名中的名（first name）。

```
2> {_,{_,Who,_},_} = Person.
{person,{name,joe,armstrong},{footsize,42}}
```

最后来看看Who的值。

```
3> Who.
joe
```

请注意，我们在前面这个例子中将_作为占位符，用于表示不感兴趣的那些变量。符号_被称为*匿名变量*。与正规变量不同，同一模式里的多个_不必绑定相同的值。

3.7　列表

列表（list）被用来存放任意数量的事物。创建列表的方法是用中括号把列表元素括起来，并用逗号分隔它们。

假设想要表示一个图形。如果假定此图形由三角形和正方形组成，就可以用一个列表来表示它。

```
1> Drawing =  [{square,{10,10},10}, {triangle,{15,10},{25,10},{30,40}},
 ...]
```

这个图形列表里的每一个元素都是固定大小的元组（例如{square, Point, Side}或者{triangle, Point1, Point2, Point3}），但这个图形本身可能包含任意数量的事物，因此用列表来表示。

列表里的各元素可以是任何类型，因此可以编写下面这个例子：

```
2> [1+7,hello,2-2,{cost, apple, 30-20},3].
[8,hello,0,{cost,apple,10},3]
```

3.7.1　专用术语

列表的第一个元素被称为列表头（head）。假设把列表头去掉，剩下的就被称为列表尾（tail）。

举个例子，如果有一个列表[1,2,3,4,5]，那么列表头就是整数1，列表尾则是列表[2,3,4,5]。请注意列表头可以是任何事物，但列表尾通常仍然是个列表。

访问列表头是一种非常高效的操作，因此基本上所有的列表处理函数都从提取列表头开始，然后对它做一些操作，接着处理列表尾。

3.7.2 定义列表

如果T是一个列表，那么[H|T]也是一个列表，它的头是H，尾是T。竖线（|）把列表的头与尾分隔开。[]是一个空列表。

LISP程序员请注意：[H|T]是一个CAR为H、CDR为T的CONS单元。此语法用在模式里会解析出CAR和CDR，用在表达式里则会构建一个CONS单元。

无论何时，只要用[...|T]语法构建一个列表，就应该确保T是列表。如果它是，那么新列表就是"格式正确的"。如果T不是列表，那么新列表就被称为"不正确的列表"。大多数库函数假定列表有正确的形式，无法用于不正确的列表。

可以给T的开头添加不止一个元素，写法是[E1,E2,..,En|T]。

举个例子，如果一开始定义ThingsToBuy如下：

```
3> ThingsToBuy = [{apples,10},{pears,6},{milk,3}].
{apples,10},{pears,6},{milk,3}]
```

那么就可以这样扩展列表：

```
4> ThingsToBuy1 = [{oranges,4},{newspaper,1}|ThingsToBuy].
[{oranges,4},{newspaper,1},{apples,10},{pears,6},{milk,3}]
```

3.7.3 提取列表元素

和其他情况一样，我们可以用模式匹配操作来提取某个列表里的元素。如果有一个非空列表L，那么表达式[X|Y] = L（X和Y都是未绑定变量）会提取列表头作为X，列表尾作为Y。

当我们在商店里，手上拿着购物单ThingsToBuy1时，所做的第一件事就是把这个列表拆成头和尾。

```
5> [Buy1|ThingsToBuy2] = ThingsToBuy1.
[{oranges,4},{newspaper,1},{apples,10},{pears,6},{milk,3}]
```

操作成功，绑定如下：Buy1 = {oranges,4}，ThingsToBuy2 = [{newspaper,1}, {apples,10}, {pears,6}, {milk,3}]。于是我们先去买橙子（oranges），然后可以继续拆出下一对商品。

```
6> [Buy2,Buy3|ThingsToBuy3] = ThingsToBuy2.
[{newspaper,1},{apples,10},{pears,6},{milk,3}]
```

操作成功后Buy2 = {newspaper,1}，Buy3 = {apples,10}，而ThingsToBuy3 = [{pears,6},{milk,3}]。

3.8 字符串

严格来说，Erlang里没有字符串。要在Erlang里表示字符串，可以选择一个由整数组成的列

表或者一个二进制型（详细请参见7.1节）。当字符串表示为一个整数列表时，列表里的每个元素都代表了一个Unicode代码点（codepoint）。

可以用字符串字面量来创建这样一个列表。字符串字面量（string literal）其实就是用双引号（"）围起来的一串字符。比如，我们可以这么写：

```
1> Name = "Hello".
"Hello"
```

"Hello"其实只是一个列表的简写，这个列表包含了代表字符串里各个字符的整数字符代码。

注意　*在一些编程语言里，字符串可以用单引号或双引号定界。而在Erlang里，必须使用双引号。*

当shell打印某个列表的值时，如果列表内的所有整数都代表可打印字符，它就会将其打印成字符串字面量。否则，打印成列表记法（有关字符集的问题请参见8.8节）。

```
2> [1,2,3].
[1,2,3]
3> [83,117,114,112,114,105,115,101].
"Surprise"
4> [1,83,117,114,112,114,105,115,101].
[1,83,117,114,112,114,105,115,101].
```

在表达式2里，列表[1,2,3]在打印时未做转换。这是因为1、2和3不是可打印字符。

在表达式3里，列表里的所有项目都是可打印字符，因此它被打印成字符串字面量。

表达式4和表达式3差不多，区别在于这个列表以1开头，而1不是可打印字符。因此，这个列表在打印时未做转换。

不需要知道代表某个字符的是哪一个整数，可以把“美元符号语法”用于这个目的。举个例子，$a实际上就是代表字符a的整数，以此类推。

```
5> I = $s.
115
6> [I-32,$u,$r,$p,$r,$i,$s,$e].
"Surprise"
```

用列表来表示字符串时，它里面的各个整数都代表Unicode字符。必须使用特殊的语法才能输入某些字符，在打印列表时也要选择正确的格式惯例。通过下面这个例子可以更好地理解这一点。

```
1> X = "a\x{221e}b".
[97,8734,98].
2> io:format("~ts~n",[X]).
a∞b
```

在第1行里，我们创建了一个包含三个整数的列表。第一个整数97是字符a的ASCII和Unicode编码。\x{221e}这种记法的作用是输入一个代表Unicode无穷大字符的十六进制整数（8734）。最后，98是字符b的ASCII和Unicode编码。shell用列表记法（[97,8734,98]）将它打印出来，这是因为8734不是一个可打印的Latin1字符编码。在第2行里，我们用一个格式化I/O语句打印出这

个字符串，里面使用了代表无穷大字符的正确字符图案。

如果shell将某个整数列表打印成字符串，而你其实想让它打印成一列整数，那就必须使用格式化的写语句，就像下面这样：

```
1> X = [97,98,99].
"abc"
2> io:format("~w~n",["abc"]).
[97,98,99]
```


3.9　模式匹配再探

为了完成这一章，我们将再次回到模式匹配上来。

下面这张表里有一些模式与单位的例子，模式里的所有变量都假定未绑定过。单位（term）就是指任何一种Erlang数据结构。表格的第二列（标记为结果）展示了模式与单位是否匹配，以及匹配时会创建哪些变量绑定。请仔细阅读这些例子，确保真正理解了它们。

模式 = 单元	结　果
{X,abc} = {123,abc}	成功：X = 123
{X,Y,Z} = {222,def,"cat"}	成功：X = 222，Y = def，Z = "cat"
{X,Y} = {333,ghi,"cat"}	失败：元组的形状不同
X = true	成功：X = true
{X,Y,X} = {{abc,12},42,{abc,12}}	成功：X = {abc,12}，Y = 42
{X,Y,X} = {{abc,12},42,true}	失败：X不能既是{abc,12}又是true
[H|T] = [1,2,3,4,5]	成功：H = 1，T = [2,3,4,5]
[H|T] = "cat"	成功：H = 99，T = "at"
[A,B,C|T] = [a,b,c,d,e,f]	成功：A = a，B = b，C = c，T = [d,e,f]

如果对其中任何一条不够确定，可以试着在shell里输入一条“模式=单位”表达式来看看会发生什么。

这里有一个例子：

```
1> {X, abc} = {123, abc}.
{123,abc}.
2> X.
123
3> f().
ok
4> {X,Y,Z} = {222,def,"cat"}.
{222,def,"cat"}.
5> X.
222
6> Y.
def
...
```

注意　`f()`命令让shell忘记现有的任何绑定。在这个命令之后，所有变量都会变成未绑定状态，因此第4行的X与第1行和第2行的X没有任何关系。

现在我们已经熟悉了基本的数据类型，以及一次性赋值和模式匹配这些概念。接下来可以加快节奏，看看如何定义模块和函数了，这就是第4章的主题。

3.10　练习

(1) 快速浏览3.1.3节，然后测试并记忆这些行编辑命令。

(2) 在shell里输入`help()`命令。你将看到一长串命令。可以试一试其中一些命令。

(3) 试着用一个元组来表示一座房子，再用一个房子列表来表示一条街道。请确保你能向这些结构中加入数据或从中取出数据。

第 4 章

模块与函数

模块和函数是构建顺序与并行程序的基本单元。模块包含了函数，而函数可以顺序或并行运行。

本章建立在上一章模式匹配概念的基础之上，介绍了编写代码所需的全部控制语句。我们将讨论高阶函数（称为fun）及如何用它们创建你自己的控制抽象。另外，还将讨论列表推导、关卡、记录和case表达式，并展示如何将它们用在代码片段里。

让我们开始吧！

4.1 模块是存放代码的地方

模块是Erlang的基本代码单元。模块保存在扩展名为.erl的文件里，而且必须先编译才能运行模块里的代码。编译后的模块以.beam作为扩展名。

编写第一个模块之前，先来回忆一下模式匹配。我们要做的就是创建一对数据结构，分别代表一个长方形和一个正方形。然后将拆开这些数据结构，提取出长方形和正方形的边长。以下是具体做法：

```
1> Rectangle = {rectangle, 10, 5}.
{rectangle, 10, 5}.
2> Square = {square, 3}.
{square, 3}
3> {rectangle, Width, Height} = Rectangle.
{rectangle,10,5}
4> Width.
10
5> Height.
5
6> {square, Side} = Square.
{square,3}
7> Side.
3
```

我们在第1行和第2行里创建了长方形（Rectangle）和正方形（Square）。第3行和第6行用模式匹配提取出长方形和正方形中的字段。第4、5和7行打印出由模式匹配表达式创建的变量绑定。在第7行之后，shell里的变量绑定是`Width = 10`、`Height = 5`和`Side = 3`。

从shell里的模式匹配到函数里的模式匹配只需要很小的一步。让我们从一个名为area的函数开始，它将计算长方形和正方形的面积。我们会把它放入一个名为geometry（几何）的模块里，并把这个模块保存在名为geometry.erl的文件里。整个模块看起来就像这样：

geometry.erl

```
-module(geometry).
-export([area/1]).

area({rectangle, Width, Height}) -> Width * Height;
area({square, Side})             -> Side * Side.
```

文件的第一行是模块声明。声明里的模块名必须与存放该模块的主文件名相同。

第二行是导出声明。Name/N这种记法是指一个带有N个参数的函数Name，N被称为函数的元数（arity）。export的参数是由Name/N项目组成的一个列表。因此，-export([area/1])的意思是带有一个参数的函数area可以在此模块之外调用。

未从模块里导出的函数只能在模块内调用。已导出函数就相当于面向对象编程语言（OOPL）里的公共方法，未导出函数则相当于OOPL里的私有方法。

area函数有两个子句。这些子句由一个分号隔开，最后的子句以句号加空白结束。每条子句都有一个头部和一个主体，两者用箭头（->）分隔。头部包含一个函数名，后接零个或更多个模式，主体则包含一列表达式（8.13节对表达式进行了定义），它们会在头部里的模式与调用参数成功匹配时执行。这些子句会根据它们在函数定义里出现的顺序进行匹配。

注意之前用于shell示例里的模式是如何成为area函数定义的一部分的。每个模式都精确对应一条子句。area函数的第一条子句：

```
area({rectangle, Width, Height}) -> Width * Height;
```

告诉我们如何计算长方形的面积。执行函数geometry:area({rectangle, 10, 5})时，area/1的第一个子句匹配了以下绑定：Width = 10和Height = 5。匹配之后，箭头->后面的代码会被执行，也就是Width * Height，即10*5等于50。请注意此函数没有显式的返回语句。它的返回值就是子句主体里最后一条表达式的值。

现在编译这个模块，然后运行它。

```
1> c(geometry).
{ok,geometry}
2> geometry:area({rectangle, 10, 5}).
50
3> geometry:area({square, 3}).
9
```

在第1行给出了命令c(geometry)，它的作用是编译geometry.erl文件里的代码。编译器返回了{ok,geometry}，意思是编译成功，而且geometry模块已被编译和加载。编译器会在当前目录创建一个名为geometry.beam的目标代码模块。在第2行和第3行调用了geometry模块里的函数。请注意，需要给函数名附上模块名，这样才能准确标明想调用的是哪个函数。

4.1.1　常见错误

警告：之前用到的c(geometry).等命令只能在shell里工作，不能放入模块。有些读者错误地把源代码清单里的代码片段输入到shell中。它们不是有效的shell命令，如果试图这样做会得到一些非常奇怪的错误消息。所以，别这么干。

如果你碰巧选择了与系统模块相冲突的模块名，那么编译模块时会得到一条奇怪的消息，说不能加载位于固定目录（sticky directory）的模块。只需重命名那个模块，然后删除编译模块时生成的所有.beam文件就可以了。

4.1.2　目录和代码路径

如果你下载了本书的代码示例或者想编写自己的示例，请务必确保在shell里运行编译器时位于正确的目录，这样系统才能找到你的文件。

Erlang shell有许多内建命令可供查看和修改当前的工作目录。

- pwd()打印当前工作目录。
- ls()列出当前工作目录里所有的文件名。
- cd(Dir)修改当前工作目录至Dir。

4.1.3　给代码添加测试

在这个阶段，可以给模块添加一些简单的测试。把模块重命名为geometry1.erl，然后添加一些测试代码。

geometry1.erl

```
-module(geometry1).
-export([test/0, area/1]).

test() ->
    12  = area({rectangle, 3, 4}),
    144 = area({square, 12}),
    tests_worked.

area({rectangle, Width, Height}) -> Width * Height;
area({square, Side})             -> Side * Side.
```

```
1> c(geometry1).
{ok,geometry1}
2> geometry1:test().
tests_worked
```

12 = area({rectangle, 3, 4})这行代码是一项测试。如果area({rectangle, 3, 4})没有返回12，模式匹配就会失败，我们会得到一条错误消息。执行geometry1:test()并看到结果是tests_worked时，就可以确定test/0主体里的所有测试都成功通过了。

无需任何额外的工具就能轻松添加测试并实施测试驱动的开发,所需的仅仅是模式匹配和=。虽然这对随手测试来说足够了，但生产用代码最好还是使用一套全功能的测试框架，比如通用或单元测试框架。详情请参阅Erlang文档①的测试部分。

4.1.4　扩展程序

现在假设想要扩展这个程序，给几何对象添加一个圆。可以这么写：

```
area({rectangle, Width, Height}) -> Width * Height;
area({square, Side})             -> Side * Side;
area({circle, Radius})           -> 3.14159 * Radius * Radius.
```

或者这么写：

```
area({rectangle, Width, Height}) -> Width * Height;
area({circle, Radius})           -> 3.14159 * Radius * Radius;
area({square, Side})             -> Side * Side.
```

请注意，这个例子里的子句顺序无关紧要。无论子句如何排列，程序都是一个意思，因为子句里的各个模式是互斥的。这让编写和扩展程序变得非常简单：只要添加更多的模式就行了。不过一般来说，子句的顺序还是很重要的。当某个函数执行时，子句与调用参数进行模式匹配的顺序就是它们在文件里出现的顺序。

在进一步讨论之前，你应当注意area函数编写方式的下列细节。

- area函数包含了多个不同的子句。当我们调用这个函数时，会从第一个与调用参数相匹配的子句开始执行。
- 我们的函数并不处理模式匹配失败的情形，程序会以一个运行时错误结束。这是有意而为的，是在Erlang里编程的方式。

在许多编程语言（例如C）里，一个函数只有一个入口点。如果是用C编写的程序，代码可能会是这样：

```
enum ShapeType { Rectangle, Circle, Square };

struct Shape {
    enum ShapeType kind;

    union {
        struct { int width, height; } rectangleData;
        struct { int radius; }        circleData;
        struct { int side;}           squareData;
    } shapeData;
};
```

① http://www.erlang.org/doc

```
double area(struct Shape* s) {
    if( s->kind == Rectangle ) {
      int width, ht;
      width = s->shapeData.rectangleData.width;
      ht    = s->shapeData.rectangleData.height;
      return width * ht;
    } else if ( s->kind == Circle ) {
      ...
```

这段C代码基本上就是对函数的参数执行模式匹配操作，但程序员必须编写模式匹配的代码，并确保它是正确的。

而在对应的Erlang代码里，我们只需要编写模式，Erlang编译器就会生成最佳的模式匹配代码，用它来选择正确的程序入口点。

下面展示了对应的Java代码：

```
abstract class Shape {
    abstract double area();
}

class Circle extends Shape {
    final double radius;
    Circle(double radius) { this.radius = radius; }
    double area() { return Math.PI * radius*radius;
}

class Rectangle extends Shape {
    final double ht;
    final double width;
    Rectangle(double width, double height) {
        this.ht = height;
        this.width  = width;
    }

    double area() { return width * ht; }
}

class Square extends Shape {
    final double side;
    Square(double side) {
        this.side = side;
    }

    double area() { return side * side; }
}
```

如果对比Erlang和Java的代码，你会看到Java程序里的`area`代码分为三处。而在Erlang程序里，所有的`area`代码都在同一个地方。

4.1.5　分号放哪里

离开geometry示例之前，我们最后再看一下它的代码，这一次看的是标点符号。请仔细观察逗号、分号和句号在代码里的位置。

geometry.erl

```
-module(geometry).
-export([area/1]).

area({rectangle, Width, Height}) -> Width * Height;
area({square, Side})             -> Side * Side.
```

你会看到以下内容。

- 逗号（,）分隔函数调用、数据构造和模式中的参数。
- 分号（;）分隔子句。我们能在很多地方看到子句，例如函数定义，以及case、if、try..catch和receive表达式。
- 句号（.）（后接空白）分隔函数整体，以及shell里的表达式。

有一种简单的方法可以记住这些：想想英语。句号分隔句子，分号分隔子句，逗号则分隔下级子句。逗号象征短程，分号象征中程，句号则象征长程。

每当我们看见一组组后接表达式的模式，就会看到它们用分号作为间隔符。这里有一个例子：

```
case f(...) of
    Pattern1 ->
       Expressions1;
    Pattern2 ->
       Expressions2;

    ...

    LastPattern ->
       LastExpression
end
```

请注意，最后的表达式（即关键字end之前的那条）没有分号。

理论说得够多了，让我们继续来看代码，过些时候再回到控制结构上来。

4.2　继续购物

在3.7.2节里，有一个像这样的购物列表：

```
[{oranges,4},{newspaper,1},{apples,10},{pears,6},{milk,3}]
```

现在假设我们想要知道购物花了多少钱。要计算它，需要知道购物列表里每一项的价格。假设此信息将在一个名为shop的模块中计算，它的定义如下：

shop.erl

```
-module(shop).
-export([cost/1]).

cost(oranges)   -> 5;
cost(newspaper) -> 8;
cost(apples)    -> 2;
cost(pears)     -> 9;
cost(milk)      -> 7.
```

函数cost/1由5个子句组成。每个子句的头部都包含一个模式（在这个案例里只是一个非常简单的原子）。当执行shop:cost(X)时，系统会尝试将X与各子句中的模式相匹配。如果发现了匹配，就会执行->右侧的代码。

来测试一下。我们将在Erlang shell里编译并运行这个程序。

```
1> c(shop).
{ok,shop}
2> shop:cost(apples).
2
3> shop:cost(oranges).
5
4> shop:cost(socks).
** exception error: no function clause matching shop:cost(socks)
   (shop.erl, line 4)
```

我们在第1行编译了shop.erl文件里的模块。在第2行和第3行，查询了apples和oranges的价格（结果中的2和5是单价）。在第4行查询了socks（袜子）的价格，但因为没有子句与其匹配，所以得到了一个模式匹配错误，系统打印出一条包含文件名和出错行号的错误消息。

再回到购物列表。假设有一个像这样的购物列表：

```
1> Buy = [{oranges,4}, {newspaper,1}, {apples,10}, {pears,6}, {milk,3}].
[{oranges,4},{newspaper,1},{apples,10},{pears,6},{milk,3}]
```

而我们想要计算列表里所有项目的总价值。一种做法是定义一个shop1:total/1函数如下：

shop1.erl

```
-module(shop1).
-export([total/1]).

total([{What, N}|T]) -> shop:cost(What) * N + total(T);
total([])            -> 0.
```

拿它试验一下：

```
2> c(shop1).
{ok,shop1}
3> shop1:total([]).
0
```

这里返回0是因为total/1的第二个子句是total([]) -> 0。

下面是一条更复杂的查询：

```
4> shop1:total([{milk,3}]).
21
```

这里的工作方式如下。shop1:total([{milk,3}])与下面这个子句相匹配，从而绑定了What = milk、N = 3以及T = []：

```
total([{What,N}|T]) -> shop:cost(What) * N + total(T);
```

接下来，函数的主体代码被执行，所以要执行的表达式是：

```
shop:cost(milk) * 3 + total([]);
```

shop:cost(milk)等于7，total([])等于0，因此最终的返回值是21。

还可以用更复杂的参数来测试它。

```
5> shop1:total([{pears,6},{milk,3}]).
75
```

同样，第5行与total/1的第一个子句相匹配，绑定了What = pears、N = 6以及T = [{milk,3}]。

```
total([{What,N}|T]) -> shop:cost(What) * N + total(T);
```

变量What、N和T在子句主体中被替换，执行的是shop:cost(pears) * 6 + total([{milk,3}])，简化后就是9 * 6 + total([{milk,3}])。

而之前已经算出total([{milk,3}])等于21，因此最终的结果是9 * 6 + 21 = 75。

最后：

```
6> shop1:total(Buy).
123
```

结束本节之前，应该更仔细地观察一下total函数。total(L)的工作方式是对参数L进行情况分析。可能的情况有两种：L要么是一个非空列表，要么是一个空列表。我们为每一种可能的情况编写一个子句，就像这样：

```
total([Head|Tail]) ->
    some_function_of(Head)  + total(Tail);
total([]) ->
    0.
```

在这个案例中，Head是模式{What,N}。当子句1匹配了一个非空列表，就会从中取出列表头，用它做一些事，然后调用自身继续处理列表尾。当列表缩减至空列表（[]）时，就会匹配子句2。

total/1函数实际上做了两件事。它首先查询列表里每个元素的价格，然后将它们乘以购买数量后的值加在一起。可以换一种方式重写total，将查找各个项目的值与累加这些值区分开来。这样代码就会更加清晰，也更易于理解。要做到这一点，我们将编写两个处理列表的小函数，分

别名为sum和map。要编写map，必须先介绍fun的概念。在此之后，将编写一个改进版的total，你可以在4.4节的shop2.erl模块里找到它。

4.3 fun：基本的抽象单元

Erlang是一种函数式编程语言。此外，函数式编程语言还表示函数可以被用作其他函数的参数，也可以返回函数。操作其他函数的函数被称为*高阶函数*（higher-order function），而在Erlang中用于代表函数的数据类型被称为fun。

高阶函数是函数式编程语言的精髓——函数式程序不仅可以操作常规的数据结构，还可以操作改变数据的函数。一旦你学会如何使用它们，就会爱上它们。在后面会看到更多的高阶函数。

可以通过下列方式使用fun。

- 对列表里的每一个元素执行相同的操作。在这个案例里，将fun作为参数传递给lists:map/2和lists:filter/2等函数。fun的这种用法是极其普遍的。
- 创建自己的控制抽象。这一技巧极其有用。例如，Erlang没有for循环，但我们可以轻松创建自己的for循环。创建控制抽象的优点是可以让它们精确实现我们想要的做法，而不是依赖一组预定义的控制抽象，因为它们的行为可能不完全是我们想要的。
- 实现可重入解析代码（reentrant parsing code）、解析组合器（parser combinator）或惰性求值器（lazy evaluator）等事物。在这个案例里，我们编写返回fun的函数。这种技术很强大，但可能会导致程序难以调试。

funs是“匿名的”函数。这样称呼它们是因为它们没有名字。你可能会看到其他编程语言称它们为lambda抽象。下面开始试验，首先将定义一个fun并将它指派给一个变量。

```
1> Double = fun(X) -> 2*X end.
#Fun<erl_eval.6.56006484>
```

定义一个fun后，Erlang shell打印出#Fun<...>，里面的...是一些古怪的数字。现在无需担心它们。

我们只能对fun做一件事，那就是给它应用一个参数，就像这样：

```
2> Double(2).
4
```

fun可以有任意数量的参数。可以编写一个函数来计算直角三角形的斜边，就像这样：

```
3> Hypot = fun(X, Y) -> math:sqrt(X*X + Y*Y) end.
#Fun<erl_eval.12.115169474>
4> Hypot(3,4).
5.0
```

如果参数的数量不正确，将会得到一个错误。

```
5> Hypot(3).
** exception error: interpreted function with arity 2 called with one argument
```

这段错误消息说明Hypot接受两个参数，但我们只提供了一个。之前说过，元数是一个函数接受的参数数量。

fun可以有多个不同的子句。这里有一个转换华氏与摄氏温度的函数：

```
6> TempConvert = fun({c,C}) -> {f, 32 + C*9/5};
6>                  ({f,F}) -> {c, (F-32)*5/9}
6>               end.
#Fun<erl_eval.6.56006484>
7> TempConvert({c,100}).
{f,212.0}
8> TempConvert({f,212}).
{c,100.0}
9> TempConvert({c,0}).
{f,32.0}
```

注意 第6行的表达式占据了多行。输入这个表达式时，每输入一个新行，shell就会重复6>这个提示符，意思是这个表达式还不完整，shell想得到更多的输入。

4.3.1 以fun作为参数的函数

标准库里的lists模块导出了一些以fun作为参数的函数。它们之中最有用的是lists:map(F, L)。这个函数返回的是一个列表，它通过给列表L里的各个元素应用fun F生成。

```
10> L = [1,2,3,4].
[1,2,3,4]
11> lists:map(fun(X) -> 2*X end, L).
[2,4,6,8]
```

另一个有用的函数是lists:filter(P, L)，它返回一个新的列表，内含L中所有符合条件的元素（条件是对元素E而言P(E)为true）。

定义一个函数Even(X)，如果X是偶数就返回true。

```
12> Even = fun(X) -> (X rem 2) =:= 0 end.
#Fun<erl_eval.6.56006484>
```

这里的X rem 2会计算出X除以2后的余数，=:=用来测试是否相等。现在可以测试Even，然后将它用作map和filter的参数了。

```
13> Even(8).
true
14> Even(7).
false
15> lists:map(Even, [1,2,3,4,5,6,8]).
[false,true,false,true,false,true,true]
16> lists:filter(Even, [1,2,3,4,5,6,8]).
[2,4,6,8]
```

map和filter等函数能在一次调用里对整个列表执行某种操作，我们把它们称为一次一列表（list-at-a-time）式操作。使用一次一列表式操作让程序变得更小，而且易于理解。之所以易于理解是因为我们可以把对整个列表的每一次操作看作程序的单个概念性步骤。否则，就必须将对列表元素的每一次操作视为程序的单独步骤了。

4.3.2　返回fun的函数

函数不仅可以使用fun作为参数（例如map和filter），还可以返回fun。

这里有一个例子。假设我有一个包含某种事物的列表，比如说水果：

```
1> Fruit = [apple,pear,orange].
[apple,pear,orange]
```

现在我可以定义一个MakeTest(L)函数，将事物列表（L）转变成一个测试函数，用来检查它的参数是否在列表L中。

```
2> MakeTest = fun(L) -> (fun(X) -> lists:member(X, L) end) end.
#Fun<erl_eval.6.56006484>
3> IsFruit = MakeTest(Fruit).
#Fun<erl_eval.6.56006484>
```

如果X是列表L中的成员，lists:member(X, L)就返回true，否则返回false。构建完测试函数之后，我们可以来试试它。

```
4> IsFruit(pear).
true
5> IsFruit(apple).
true
6> IsFruit(dog).
false
```

也可以把它用作lists:filter/2的参数。

```
7> lists:filter(IsFruit, [dog,orange,cat,apple,bear]).
[orange,apple]
```

这种用fun来返回fun的写法可能需要一些时间才能习惯，所以先来分解一下这种写法，让其中的过程更清楚一些。一个返回“正常”值的函数是这样的：

```
1> Double = fun(X) -> ( 2 * X ) end.
#Fun<erl_eval.6.56006484>
2> Double(5).
10
```

括号里的代码（也就是2 * X）很明显就是此函数的“返回值”。现在试着把一个fun放到括号内。

记住，括号里的东西*就是*返回值。

```
3> Mult = fun(Times) -> ( fun(X) -> X * Times end ) end.
#Fun<erl_eval.6.56006484>
```

括号里的fun是fun(X) -> X * Times end，它只是一个关于X的函数。Times是“外部”fun的参数。

Mult(3)执行后返回fun(X) -> X * 3 end，即内部fun的主体（Times被替换为3）。现在我们可以测试它了。

```
4> Triple = Mult(3).
#Fun<erl_eval.6.56006484>
5> Triple(5).
15
```

所以，Mult是通用化的Double。它不计算值，而是返回一个函数，调用该函数时会计算所需的值。

4.3.3　定义你自己的控制抽象

到目前为止，还没有看到任何的if语句、switch语句、for语句或while语句，然而这似乎没什么问题。所有的一切都是用模式匹配和高阶函数编写的。

如果想要额外的控制结构，可以自己创建。这里有个例子，Erlang没有for循环，所以我们来创建一个：

lib_misc.erl

```
for(Max, Max, F) -> [F(Max)];
for(I, Max, F)   -> [F(I)|for(I+1, Max, F)].
```

举个例子，执行for(1,10,F)会创建列表[F(1), F(2), ..., F(10)]。

现在已经有了一个简单的for循环。我们可以用它生成一个从1到10的整数列表。

```
1> lib_misc:for(1,10,fun(I) -> I end).
[1,2,3,4,5,6,7,8,9,10]
```

或者，也可以计算从1到10的整数平方。

```
2> lib_misc:for(1,10,fun(I) -> I*I end).
[1,4,9,16,25,36,49,64,81,100]
```

等你有了经验，就会发现创建自己的控制结构能大大降低程序的大小，有时还能让它们更加清晰。这是因为你能精确地创建出解决问题所需要的控制结构，同时还不受编程语言自带的少量固定控制结构所限。

4.4　简单列表处理

介绍完fun之后，可以继续编写sum和map了。我们需要用它们来实现改进版的total（相信你还没有忘记它）。

先从sum开始，它计算某个列表里所有元素的总和。[①]

mylists.erl

```
❶ sum([H|T]) -> H + sum(T);
❷ sum([])    -> 0.
```

注意，这两个sum子句的顺序并不重要。因为子句1匹配一个非空列表，而子句2匹配空列表，这两种情况是互斥的。可以像下面这样测试sum：

```
1> c(mylists).  %% <-- Last time I do this
{ok, mylists}
2> L = [1,3,10].
[1,3,10]
3> mylists:sum(L).
14
```

4

第1行编译了mylists模块。从现在起，我会不时地省略编译模块的命令，因此你必须记得自己做这一步。这一点也不难理解。下面来跟踪一下执行过程。

(1) sum([1,3,10])

(2) sum([1,3,10]) = 1 + sum([3,10]) （通过❶）

(3) = 1 + 3 + sum([10]) （通过❶）

(4) = 1 + 3 + 10 + sum([]) （通过❶）

(5) = 1 + 3 + 10 + 0 （通过❷）

(6) = 14

最后，来看看前面曾遇到的map/2。它的定义如下：

mylists.erl

```
❶ map(_, [])     -> [];
❷ map(F,  [H|T]) -> [F(H)|map(F, T)].
```

❶ 子句1说明如何处理空列表。让任何函数映射一个空列表里的元素（一个也没有！）只会得到一个空列表。

❷ 子句2规定了如何处理带有头H和尾T的列表。这很简单，只要构建一个头为F(H)、尾为map(F, T)的新列表就可以了。

注意 map/2的定义是直接从标准库的lists模块复制到mylists里的。你可以对mylists.erl里的代码做任何想做的事,但在任何情况下都不要试图制作自己的lists模块——在lists里犯的任何错误都可能会给系统造成严重损坏。

可以用map来运行一些函数，得到某个列表里所有元素的双倍或平方，就像下面这样：

① 代码片段中的❶❷在下面会用到，下同。——译者注

```
1> L = [1,2,3,4,5].
[1,2,3,4,5]
2> mylists:map(fun(X) -> 2*X end, L).
[2,4,6,8,10]
3> mylists:map(fun(X) -> X*X end, L).
[1,4,9,16,25]
```

稍后会展示map的进一步简化版，用列表推导编写。26.3节会展示如何并行计算所有的映射元素（这样在多核计算机上就会加速我们的程序），不过这个话题太过超前了。现在我们已经了解了sum和map，可以用这两个函数重写total了：

shop2.erl

```
-module(shop2).
-export([total/1]).
-import(lists, [map/2, sum/1]).

total(L) ->
    sum(map(fun({What, N}) -> shop:cost(What) * N end, L)).
```

可以通过观察其中的步骤来了解这个函数的工作方式。

```
1> Buy = [{oranges,4},{newspaper,1},{apples,10},{pears,6},{milk,3}].
[{oranges,4},{newspaper,1},{apples,10},{pears,6},{milk,3}]
2> L1=lists:map(fun({What,N}) -> shop:cost(What) * N end, Buy).
[20,8,20,54,21]
3> lists:sum(L1).
123
```

我是如何编写程序的

编写程序时，我的做法是“编写一点”然后“测试一点”。我从一个包含少量函数的小模块开始，先编译它，然后在shell里用一些命令测试它。当我觉得满意后，就会再编写一些函数，编译它们，测试它们，以此类推。

通常，我并不完全确定程序需要什么样的数据结构，通过测试那些样本代码，我就能看出所选的数据结构是否合适。

我倾向于让程序“生长”，而不是事先就完全想好要如何编写它们，这样就不会在出现问题时才发现犯了大错。最重要的是，这样做很有趣。我能立即获得反馈，而且只要在程序里输入就能知道我的想法是否有效。

一旦弄清楚如何在shell里做某些事情，我通常就会转而编写makefile和一些代码，重现我在shell里学到的内容。

还应该注意模块里-import和-export声明的用法。

- -import(lists, [map/2, sum/1]).声明的意思是map/2函数是从lists模块里导入的，后面的也一样。这就意味着我们可以用map(Fun, ...)来代替lists:map(Fun, ...)的

写法了。cost/1没有在导入声明中声明过，因此必须使用“完全限定”（fully qualified）的名称shop:cost。

- -export([total/1])声明的意思是total/1函数可以在shop2模块之外调用。只有从一个模块里导出的函数才能在该模块之外调用。

现在，你可能会认为total函数没有改进的空间了，那样你就错了。进一步的优化是可能的。为了做到这一点，我们将使用列表推导。

4.5　列表推导

列表推导（list comprehension）是无需使用fun、map或filter就能创建列表的表达式。它让程序变得更短，更容易理解。

我们从一个例子开始。假设有一个列表L，

```
1> L = [1,2,3,4,5].
[1,2,3,4,5]
```

并且想要让列表里的每个元素加倍。虽然之前做过，但在这里再次展示一下做法。

```
2> lists:map(fun(X) -> 2*X end, L).
[2,4,6,8,10]
```

而用列表推导的方式就简单多了。

```
4> [2*X || X <- L ].
[2,4,6,8,10]
```

[F(X) || X <- L]标记的意思是“由F(X)组成的列表（X从列表L中提取）”。因此，[2*X || X <- L]的意思就是“由2*X组成的列表（X从列表L中提取）”。

要了解如何使用列表推导，可以在shell里输入一些表达式，看看会发生什么。我们从定义Buy开始。

```
1> Buy=[{oranges,4},{newspaper,1},{apples,10},{pears,6},{milk,3}].
[{oranges,4},{newspaper,1},{apples,10},{pears,6},{milk,3}]
```

现在，把原始列表里每一项的数字加倍。

```
2> [{Name, 2*Number} || {Name, Number} <- Buy].
[{oranges,8},{newspaper,2},{apples,20},{pears,12},{milk,6}]
```

请注意，||符号右侧的元组{Name, Number}是一个模式，用于匹配列表Buy里的各个元素。左侧的元组{Name, 2*Number}则是一个构造器（constructor）。

假设想要计算原始列表里所有元素的总价，可以像下面这样做。首先将每一项的名称替换成它的价格。

```
3> [{shop:cost(A), B} || {A, B} <- Buy].
[{5,4},{8,1},{2,10},{9,6},{7,3}]
```

现在让两个数字相乘。

```
4> [shop:cost(A) *  B || {A, B} <- Buy].
[20,8,20,54,21]
```

然后把它们全部加起来。

```
5> lists:sum([shop:cost(A) *  B || {A, B} <- Buy]).
123
```

最后，如果想把它做成一个函数，就会像下面这样写：

```
total(L) ->
    lists:sum([shop:cost(A) *  B || {A, B} <- L]).
```

列表推导会让代码变得非常短并且易于阅读。举个例子，可以定义一个进一步简化的map。

```
map(F, L) -> [F(X) || X <- L].
```

列表推导最常规的形式是下面这种表达式：

```
[X || Qualifier1, Qualifier2, ...]
```

X是任意一条表达式，后面的限定符（Qualifier）可以是生成器、位串生成器或过滤器。

- 生成器（generator）的写法是Pattern <- ListExpr，其中的ListExp必须是一个能够得出列表的表达式。
- 位串（bitstring）生成器的写法是BitStringPattern <= BitStringExpr，其中的BitStringExpr必须是一个能够得出位串的表达式。更多有关位串模式和生成器的信息请参阅Erlang参考手册[①]。
- 过滤器（filter）既可以是判断函数（即返回true或false的函数），也可以是布尔表达式。

请注意，列表推导里的生成器部分起着过滤器的作用，这里有一个例子：

```
1> [ X || {a, X} <- [{a,1},{b,2},{c,3},{a,4},hello,"wow"]].
[1,4]
```

本节最后会介绍几个简短的例子。

4.5.1　Quicksort

以下是如何用两个列表推导来编写一种排序算法：

lib_misc.erl

```
qsort([]) -> [];
qsort([Pivot|T]) ->
	qsort([X || X <- T, X < Pivot])
	++ [Pivot] ++
	qsort([X || X <- T, X >= Pivot]).
```

① http://www.erlang.org/doc/pdf/otp-system-documentation.pdf

注意这里的++是中缀插入操作符。展示这段代码是为了表现它的优雅，而不是效率。这样使用++一般不认为是良好的编程实践做法。更多信息请参见4.9节。

```
1> L=[23,6,2,9,27,400,78,45,61,82,14].
[23,6,2,9,27,400,78,45,61,82,14]
2> lib_misc:qsort(L).
[2,6,9,14,23,27,45,61,78,82,400]
```

为了了解它是如何工作的，我们将一步步展示执行的过程。先从一个列表L开始，对它调用qsort(L)。下面这一步匹配qsort的子句2，产生了如下绑定：Pivot → 23和T → [6,2,9,27,400,78,45,61,82,14]：

```
3> [Pivot|T] = L.
[23,6,2,9,27,400,78,45,61,82,14]
```

现在将T分成两个列表，一个包含T里所有小于Pivot（中位数）的元素，另一个包含所有大于或等于Pivot的元素。

```
4> Smaller = [X || X <- T, X < Pivot].
[6,2,9,14]
5> Bigger  = [X || X <- T, X >= Pivot].
[27,400,78,45,61,82]
```

现在排序Smaller和Bigger，并将它们与Pivot合并。

```
qsort( [6,2,9,14] ) ++ [23] ++ qsort( [27,400,78,45,61,82] )
= [2,6,9,14] ++ [23] ++ [27,45,61,78,82,400]
= [2,6,9,14,23,27,45,61,78,82,400]
```

4.5.2 毕达哥拉斯三元数组

毕达哥拉斯三元数组[①]是由整数{A,B,C}构成的数组，其中$A^2+B^2=C^{①}$。

pythag(N)函数会生成一个包含所有整数{A,B,C}组合的列表，其中$A^2+B^2=C^2$并且各条边之和小于等于N。

lib_misc.erl

```
pythag(N) ->
    [ {A,B,C} ||
        A <- lists:seq(1,N),
        B <- lists:seq(1,N),
        C <- lists:seq(1,N),
        A+B+C =< N,
        A*A+B*B =:= C*C
    ].
```

简单解释一下：lists:seq(1, N)返回一个包含从1到N所有整数的列表。因此，A <- lists:

① 即勾股定理中的勾、股和弦。——译者注

seq(1, N)的意思是A提取从1到N的所有可能值。所以这个程序可以这样念：“提取1到N的所有*A*值，1到N的所有*B*值，1到N的所有*C*值，条件是*A* + *B* + *C*小于等于N并且A*A + B*B = C*C。”

```
1> lib_misc:pythag(16).
[{3,4,5},{4,3,5}]
2> lib_misc:pythag(30).
[{3,4,5},{4,3,5},{5,12,13},{6,8,10},{8,6,10},{12,5,13}]
```

4.5.3　回文构词

如果你对英式填字游戏感兴趣，就应该经常在琢磨回文构词（anagram）。让我们借助下面这个精致的小函数perms，用Erlang来找到一个字符串的所有排列形式。

lib_misc.erl

```
perms([]) -> [[]];
perms(L)  -> [[H|T] || H <- L, T <- perms(L--[H])].
```

```
1> lib_misc:perms("123").
["123","132","213","231","312","321"]
2> lib_misc:perms("cats").
["cats", "cast", "ctas", "ctsa", "csat", "csta", "acts", "acst",
"atcs", "atsc", "asct", "astc", "tcas", "tcsa", "tacs", "tasc",
"tsca", "tsac", "scat", "scta", "sact", "satc", "stca", "stac"]
```

X -- Y是列表移除操作符，它从X里移除Y中的元素。8.16节提供了更准确的定义。

perms十分简洁。它的工作方式如下：假设想要计算字符串"cats"的所有排列形式。首先，分离字符串的第一个字符，也就是c，并计算字符串移除c后的所有排列形式。"cats"移除c后是字符串"ats"，而"ats"的全部排列形式是以下字符串：["ats", "ast", "tas", "tsa", "sat", "sta"]。接下来，把c附加到所有这些字符串的开头，形成["cats", "cast", "ctas", "ctsa", "csat", "csta"]。然后继续分离第二个字符并重复这一算法，以此类推。

perms函数所做的就是以上这些。

```
[ [H|T] || H <- L, T <- perms(L -- [H]) ]
```

它的意思是穷尽一切可能从L里提取H，然后穷尽一切可能从perms(L -- [H])（即列表L移除H后的所有排列形式）里提取T，最后返回[H|T]。

4.6　内置函数

内置函数简称为BIF（built-in function），是那些作为Erlang语言定义一部分的函数。有些内置函数是用Erlang实现的，但大多数是用Erlang虚拟机里的底层操作实现的。

内置函数能提供操作系统的接口，并执行那些无法用Erlang编写或者编写后非常低效的操作。比如，你无法将一个列表转变成元组，或者查到当前的时间和日期。要执行这样的操作，需

要调用内置函数。

举个例子，内置函数list_to_tuple/1能将一个列表转换成元组，time/0以{时，分，秒}的格式返回当前的时间。

```
1> list_to_tuple([12,cat,"hello"]).
{12,cat,"hello"}
2> time().
{20,0,3}
```

所有内置函数都表现得像是属于erlang模块，但那些最常用的内置函数（例如list_to_tuple）是自动导入的，因此可以直接调用list_to_tuple(...)，而无需用erlang:list_to_tuple(...)。

可以在erlang手册页里找到所有内置函数的完整清单，它位于Erlang分发套装内，也可以在线查看：http://www.erlang.org/doc/man/erlang.html。在本书余下部分里，我只会介绍理解特定章节所必需的内置函数。系统中的内置函数绝不止本书里介绍的这些，因此建议你将手册页打印出来，试着去了解所有的内置函数。

4.7 关卡

关卡（guard）是一种结构，可以用它来增加模式匹配的威力。通过使用关卡，可以对某个模式里的变量执行简单的测试和比较。假设想要编写一个计算X和Y之间最大值的max(X, Y)函数。可以像下面这样用关卡来编写它：

```
max(X, Y) when X > Y -> X;
max(X, Y) -> Y.
```

子句1会在X大于Y时匹配，结果是X。

如果子句1不匹配，系统就会尝试子句2。子句2总是返回第二个参数Y。Y必然是大于或等于X的，否则子句1就已经匹配了。

可以在函数定义里的头部使用关卡（通过关键字when引入），也可以在支持表达式的任何地方使用它们。当它们被用作表达式时，执行的结果是true或false这两个原子中的一个。如果关卡的值为true，我们会说执行*成功*，否则执行*失败*。

4.7.1 关卡序列

关卡序列（guard sequence）是指单一或一系列的关卡，用分号（;）分隔。对于关卡序列G1; G2; ...; Gn，只要其中有一个关卡（G1、G2……）的值为true，它的值就为true。

关卡由一系列*关卡表达式*组成，用逗号（,）分隔。关卡GuardExpr1, GuardExpr2, ..., GuardExprN只有在所有的关卡表达式（GuardExpr1、GuardExpr2……）都为true时才为true。

合法的关卡表达式是所有合法Erlang表达式的一个子集。之所以限制关卡表达式只能是Erlang表达式子集，是想要确保关卡表达式的执行是无副作用的。关卡是模式匹配的一种扩展，

而因为模式匹配无副作用，所以我们不希望关卡的执行带有副作用。

另外，关卡不能调用用户定义的函数，因为要确保它们没有副作用并能正常结束。

下列语法形式是合法的关卡表达式：

- 原子true；
- 其他常量（各种数据结构和已绑定变量），它们在关卡表达式里都会成为false；
- 调用后面表1里的关卡判断函数和表2里的内置函数；
- 数据结构比较（参见表6）；
- 算术表达式（参见表3）；
- 布尔表达式（参见8.7节）；
- 短路布尔表达式（参见8.23节）。

注意　阅读表1和表2时你会看到里面引用了我们尚未讨论过的数据类型。它们是出于完整性的考虑而被纳入这些表格的。

关卡表达式的执行适用于8.20节里介绍的优先级规则。

4.7.2　关卡示例

我们已经讨论过关卡的语法了，它有时会变得相当复杂。这里有一些例子：

```
f(X,Y) when is_integer(X), X > Y, Y < 6 -> ...
```

它的意思是"当X是一个整数，X大于Y并且Y小于6时"。关卡里分隔各个测试的逗号的意思是"并且"。

```
is_tuple(T), tuple_size(T) =:= 6, abs(element(3, T)) > 5
element(4, X) =:= hd(L)
...
```

第一行的意思是T是一个包含六个元素的元组，并且T中第三个元素的绝对值大于5。第二行的意思是元组X的第4个元素与列表L的列表头相同。

```
X =:= dog; X =:= cat
is_integer(X), X > Y ; abs(Y) < 23
...
```

第一个关卡的意思是X是一个cat或者dog，关卡里分号（;）的意思是"或者"。第二个关卡的意思是X是一个整数，并且X大于Y或者Y的绝对值小于23。

这里还有一些关卡的例子，它们使用了短路布尔表达式：

```
A >= -1.0 andalso A+1 > B
is_atom(L) orelse (is_list(L) andalso length(L) > 2)
```

允许布尔表达式用于关卡是为了使关卡在语法上类似于其他的表达式。orelse和andalso操作符存在的原因是布尔操作符and/or原本的定义是两侧参数都需要求值。在关卡里，（and与

andalso）之间和（or与orelse）之间可能会存在差别。举个例子，请考虑下面这两个关卡：

```
f(X) when (X == 0) or (1/X > 2) ->
    ...

g(X) when (X == 0) orelse (1/X > 2) ->
    ...
```

当X为0时，f(X)里的关卡会失败，但g(X)里的关卡会成功。

在实践中，很少有程序会使用复杂的关卡，简单的（,）关卡已能满足大多数程序的要求。

4.7.3 true关卡的作用

你可能会疑惑为什么会需要true关卡。原因是原子true可以被当作“来者不拒”的关卡，放置在某个if表达式的最后，就像这样：

```
if
    Guard -> Expressions;
    Guard -> Expressions;
    ...
    true  -> Expressions
end
```

我们会在4.8.2节中讨论if。

下面这些表格列出了所有的关卡判断函数（即返回布尔值的关卡）和所有的关卡内置函数。

表1 关卡判断函数

判断函数	意 思
is_atom(X)	X是一个原子
is_binary(X)	X是一个二进制型
is_constant(X)	X是一个常量
is_float(X)	X是一个浮点数
is_function(X)	X是一个fun
is_function(X, N)	X是一个带有*N*个参数的fun
is_integer(X)	X是一个整数
is_list(X)	X是一个列表
is_map(X)	X是一个映射组
is_number(X)	X是一个整数或浮点数
is_pid(X)	X是一个进程标识符
is_pmod(X)	X是一个参数化模块的实例
is_port(X)	X是一个端
is_reference(X)	X是一个引用
is_tuple(X)	X是一个元组
is_record(X,Tag)	X是一个类型为Tag的记录
is_record(X,Tag,N)	X是一个类型为Tag、大小为*N*的记录

表2 关卡内置函数

函 数	意 思
abs(X)	X的绝对值
byte_size(X)	X的字节数，X必须是一个位串或二进制型
element(N, X)	X里的元素N，注意X必须是一个元组
float(X)	将X转换成一个浮点数，X必须是一个数字
hd(X)	列表X的列表头
length(X)	列表X的长度
node()	当前的节点
node(X)	创建X的节点，X可以是一个进程、标识符、引用或端口
round(X)	将X转换成一个整数，X必须是一个数字
self()	当前进程的进程标识符
size(X)	X的大小，它可以是一个元组或二进制型
trunc(X)	将X去掉小数部分取整，X必须是一个数字
tl(X)	列表X的列表尾
tuple_size(T)	元组T的大小

4.8 case 和 if 表达式

到目前为止，所有的问题我们都用模式匹配解决。这让Erlang代码小而一致。但是，为所有问题都单独定义函数子句有时候很不方便。这时，可以使用case或if表达式。

4.8.1 case表达式

case的语法如下：

```
case Expression of
    Pattern1 [when Guard1] -> Expr_seq1;
    Pattern2 [when Guard2] -> Expr_seq2;
    ...
end
```

case的执行过程如下：首先，Expression被执行，假设它的值为Value。随后，Value轮流与Pattern1（带有可选的关卡Guard1）、Pattern2等模式进行匹配，直到匹配成功。一旦发现匹配，相应的表达式序列就会执行，而表达式序列执行的结果就是case表达式的值。如果所有模式都不匹配，就会发生异常错误（exception）。

之前用过一个名为filter(P, L)的函数，它返回一个列表，内含L里所有满足条件的元素X（条件是P(X)为true）。可以像下面这样用case来定义filter：

```
filter(P, [H|T]) ->
    case P(H) of
        true  -> [H|filter(P, T)];
        false -> filter(P, T)
```

```
    end;
filter(P, []) ->
    [].
```

严格来说，case不是必需的。下面展示了如何只用模式匹配来定义filter：

```
filter(P, [H|T]) ->  filter1(P(H), H, P, T);
filter(P, [])    ->  [].

filter1(true, H, P, T)  -> [H|filter(P, T)];
filter1(false, H, P, T) -> filter(P, T).
```


废弃的关卡函数

如果你遇到一些编写于几年前的Erlang旧代码，里面的关卡测试名称可能会不一样。旧代码使用的关卡测试名为atom(X)、constant(X)、float(X)、integer(X)、list(X)、number(X)、pid(X)、port(X)、reference(X)、tuple(X)和binary(X)。这些测试与is_atom(X)等现代版测试的含义相同。建议不要在如今的代码里使用这些旧名称。

这个定义比较丑陋。必须创建一个额外的函数（名为filter1），并向它传递filter/2的所有参数。

4.8.2 if表达式

Erlang还提供了第二种条件句式if，语法如下：

```
if
  Guard1 ->
    Expr_seq1;
  Guard2 ->
    Expr_seq2;
  ...
end
```

它的执行过程如下：首先执行Guard1。如果得到的值为true，那么if的值就是执行表达式序列Expr_seq1所得到的值。如果Guard1不成功，就会执行Guard2，以此类推，直到某个关卡成功为止。if表达式必须至少有一个关卡的执行结果为true，否则就会发生异常错误。

很多时候，if表达式的最后一个关卡是原子true，确保当其他关卡都失败时表达式的最后部分会被执行。

有一点可能会让人困惑，就是在if表达式的最后使用true关卡。如果使用C之类的语言，编写不带else部分的if语句是可行的，就像这样：

```
if( a > 0) {
    do_this();
}
```

因此，你可能会想要在Erlang里编写下面的代码：

```
if
   A > 0 ->
       do_this()
end
```

这样做在Erlang里可能会带来问题，因为if是一种表达式，而所有的表达式都应该有值。在A小于或等于0的情况下，这个if表达式是没有值的。这在Erlang里属于错误，会导致程序崩溃，但在C里不算是错误。

为了避免可能的异常错误，Erlang程序员经常会在if表达式的最后添加一个true关卡。当然，如果他们想让异常错误生成，就会省略额外的true关卡。

4.9　构建自然顺序的列表

构建列表最有效率的方式是向某个现成列表的头部添加元素，因此经常能看到包含以下模式的代码：

```
some_function([H|T], ..., Result, ...) ->
    H1 = ... H ...,
    some_function(T, ..., [H1|Result], ...);
some_function([], ..., Result, ...) ->
    {..., Result, ...}.
```

这段代码会遍历一个列表，提取出列表头H并根据函数的算法计算出某个值（可以称之为H1），然后把H1添加到输出列表Result里。当输入列表被穷尽后，最后的子句匹配成功，函数返回输出变量Result。

Result的元素顺序和原始列表的元素顺序是相反的，这也算不上什么问题。如果它们的顺序是错误的，可以很容易在最后一步里反转过来。

它的基本概念相当简单。

(1) 总是向列表头添加元素。

(2) 从输入列表的头部提取元素，然后把它们添加到输出列表的头部，形成的结果是与输入列表顺序相反的输出列表。

(3) 如果顺序很重要，就调用lists:reverse/1这个高度优化过的函数。

(4) 避免违反以上的建议。

注意　每当想要反转一个列表时，都应该调用lists:reverse，别用其他的做法。如果查看列表模块的源代码，会发现reverse的定义。但是，这个定义仅仅是用来帮助理解的。当编译器发现对lists:reverse的调用时，会调用此函数的一个更高效的内部版本。

只要看到像下面这样的代码，就应该注意了，因为这是非常低效的，只有List很短时才勉强可用：

```
List ++ [H]
```

虽然++可能会导致低效的代码，但是在清晰度和性能之间需要进行权衡。使用++可以让程序更清晰同时也不存在性能问题。最佳做法是首先把程序编写得尽可能清晰，然后，如果有性能问题，先测量再进行优化。

4.10 归集器

我们经常会想要让一个函数返回两个列表。举个例子，我们可能会想编写一个函数，把某个整数列表一分为二，分别包含原始列表里的奇数和偶数。下面是一种做法：

lib_misc.erl

```
odds_and_evens1(L) ->
    Odds  = [X || X <- L, (X rem 2) =:= 1],
    Evens = [X || X <- L, (X rem 2) =:= 0],
    {Odds, Evens}.
```

```
5> lib_misc:odds_and_evens1([1,2,3,4,5,6]).
{[1,3,5],[2,4,6]}
```

这段代码的问题在于：遍历了列表两次。如果列表很短则问题不大，但如果列表很长，就可能是个问题。

要避免遍历列表两次，可以重写代码如下：

lib_misc.erl

```
odds_and_evens2(L) ->
    odds_and_evens_acc(L, [], []).

odds_and_evens_acc([H|T], Odds, Evens) ->
    case (H rem 2) of
        1 -> odds_and_evens_acc(T, [H|Odds], Evens);
        0 -> odds_and_evens_acc(T, Odds, [H|Evens])
    end;
odds_and_evens_acc([], Odds, Evens) ->
    {Odds, Evens}.
```

现在程序只遍历列表一次，把奇偶参数分别添加到合适的列表里。这些列表被称为*归集器*（accumulator）。这段代码还有一个不太明显的额外的优点：带归集器的版本比`[H || filter(H)]`类型结构的版本更节省空间。

如果运行这段代码，得到的结果和之前几乎一样。

```
1> lib_misc:odds_and_evens2([1,2,3,4,5,6]).
{[5,3,1],[6,4,2]}
```

区别在于：奇偶列表里的元素顺序是反转的。这是列表的构建方式所导致的结果。如果想让列表元素的顺序和最初的一致，只需在函数最后的子句里反转这些列表即可，做法是修改

odds_and_evens2的第二个子句：

```
odds_and_evens_acc([], Odds, Evens) ->
    {lists:reverse(Odds), lists:reverse(Evens)}.
```

现在你已经有足够的知识来编写和理解大量的Erlang代码了。我们讨论了模块与函数的基本结构，以及编写顺序程序所需要的大多数控制结构与编程技巧。

Erlang还有两种数据类型，名为记录和映射组。它们都被用于保存复杂的数据类型。记录的作用是给某个元组里的元素赋名。这在元组包含大量元素时很有用。记录和映射组是下一章的主题。

4.11　练习

找到erlang模块的手册页。你会看到它列出了大量的内置函数（远多于我们在这里讨论过的）。可以用这些信息来解决下面列出的一些问题。

(1) 扩展geometry.erl。添加一些子句来计算圆和直角三角形的面积。添加一些子句来计算各种几何图形的周长。

(2) 内置函数tuple_to_list(T)能将元组T里的元素转换成一个列表。请编写一个名为my_tuple_to_list(T)的函数来做同样的事，但不要使用相同功能的内置函数。

(3) 查看erlang:now/0、erlang:date/0和erlang:time/0的定义。编写一个名为my_time_func(F)的函数，让它执行fun F并记下执行时间。编写一个名为my_date_string()的函数，用它把当前的日期和时间改成整齐的格式。

(4) 高级练习：查找Python datetime模块的手册页。找出Python的datetime类里有多少方法可以通过erlang模块里有关时间的内置函数实现。在erlang的手册页里查找等价的函数。如果有明显的遗漏，就实现它。

(5) 编写一个名为math_functions.erl的模块，并导出函数even/1和odd/1。even(X)函数应当在X是偶整数时返回true，否则返回false。odd(X)应当在X是奇整数时返回true。

(6) 向math_functions.erl添加一个名为filter(F, L)的高阶函数，它返回L里所有符合条件的元素X（条件是F(X)为true）。

(7) 向math_functions.erl添加一个返回{Even, Odd}的split(L)函数，其中Even是一个包含L里所有偶数的列表，Odd是一个包含L里所有奇数的列表。请用两种不同的方式编写这个函数，一种使用归集器，另一种使用在练习6中编写的filter函数。

第 5 章

记录与映射组

到目前为止，我们已经讨论了两种数据容器，分别是*元组*和*列表*。元组用于保存固定数量的元素，而列表用于保存可变数量的元素。

本章将介绍*记录*（record）和*映射组*（map）。记录其实就是元组的另一种形式。通过使用记录，可以给元组里的各个元素关联一个名称。

映射组是键–值对的关联性集合。键可以是任意的Erlang数据类型。它们在Perl和Ruby里被称为散列（hash），在C++和Java里被称为映射（map），在Lua里被称为表（table），在Python里则被称为字典（dictionary）。

使用记录和映射组能让编程更容易。与其记住某个数据项在复杂数据结构里的存放位置，不如使用该项的名称，让系统找到数据存放的位置。记录使用一组固定且预定义的名称，而映射组可以动态添加新的名称。

5.1　何时使用映射组或记录

记录其实就是元组的另一种形式，因此它们的存储与性能特性和元组一样。映射组比元组占用更多的存储空间，查找起来也更慢。而另一方面，映射组比元组要灵活得多。

应该在下列情形里使用记录：

- 当你可以用一些预先确定且数量固定的原子来表示数据时；
- 当记录里的元素数量和元素名称不会随时间而改变时；
- 当存储空间是个问题时，典型的案例是你有一大堆元组，并且每个元组都有相同的结构。

映射组适合以下的情形：

- 当键不能预先知道时用来表示键–值数据结构；
- 当存在大量不同的键时用来表示数据；
- 当方便使用很重要而效率无关紧要时作为万能的数据结构使用；
- 用作“自解释型”的数据结构，也就是说，用户容易从键名猜出值的含义；
- 用来表示键–值解析树，例如XML或配置文件；
- 用JSON来和其他编程语言通信。

5.2 通过记录命名元组里的项

对于小型元组而言，记住各个元素代表什么几乎不成问题，但当元组包含大量元素时，给各个元素命名就更方便了。一旦命名了这些元素，就可以通过名称来指向它们，而不必记住它们在元组里的具体位置。

用记录声明来命名元组里的元素，它的语法如下：

```
-record(Name, {
               %% 以下两个键带有默认值
               key1 = Default1,
               key2 = Default2,
               ...
               %% 下一行就相当于Key 3 = undefined
               key3,
               ...

             }).
```

警告　record不是一个shell命令（在shell里要用rr，详见本节后面的描述）。记录声明只能在Erlang源代码模块里使用，不能用于shell。

在之前的例子里，Name是记录名。key1、key2这些是记录所含各个字段的名称，它们必须是原子。记录里的每个字段都可以带一个默认值，如果创建记录时没有指定某个字段的值，就会使用默认值。

举个例子，假设想要操作一个待办事项列表。我们会首先定义一个todo记录，然后将它保存在一个文件里（记录的定义既可以保存在Erlang源代码文件里，也可以由扩展名为.hrl的文件保存，然后包含在Erlang源代码文件里）。

请注意，文件包含是唯一能确保多个Erlang模块共享相同记录定义的方式。它类似于C语言用.h文件保存公共定义，然后包含在源代码文件里。有关包含命令的详细内容请参阅8.15节。

records.hrl

```
-record(todo, {status=reminder,who=joe,text}).
```

记录一旦被定义，就可以创建该记录的实例了。

要在shell里这么做，必须先把记录的定义读入shell，然后才能创建记录。我们将用shell函数rr（read records的缩写，即读取记录）来实现。

```
1> rr("records.hrl").
[todo]
```

5.2.1 创建和更新记录

现在我们已经准备好定义和操作记录了。

```
2> #todo{}.
#todo{status = reminder,who = joe,text = undefined}
3> X1 = #todo{status=urgent, text="Fix errata in book"}.
#todo{status = urgent,who = joe,text = "Fix errata in book"}
4> X2 = X1#todo{status=done}.
#todo{status = done,who = joe,text = "Fix errata in book"}
```

我们在第2行和第3行创建了新的记录。语法`#todo{key1=Val1, ..., keyN=ValN}`用于创建一个类型为`todo`的新纪录。所有的键都是原子，而且必须与记录定义里所用的一致。如果省略了一个键，系统就会用记录定义里的值作为该键的默认值。

在第4行复制了一个现有的记录。语法`X1#todo{status=done}`的意思是创建一个`X1`的副本（类型必须是`todo`），并修改字段`status`的值为`done`。请记住，这么做生成的是原始记录的一个副本，原始记录没有变化。

5

5.2.2 提取记录字段

要在一次操作中提取记录的多个字段，可以使用模式匹配。

```
5> #todo{who=W, text=Txt} = X2.
#todo{status = done,who = joe,text = "Fix errata in book"}
6> W.
joe
7> Txt.
"Fix errata in book"
```

我们在匹配操作符（=）的左侧编写了一个记录模式，包含了未绑定变量`W`和`Txt`。如果匹配成功，这些变量就会绑定记录里的相应字段。如果只是想要记录里的单个字段，就可以使用“点语法”来提取该字段。

```
8> X2#todo.text.
"Fix errata in book"
```

5.2.3 在函数里模式匹配记录

我们可以编写模式匹配记录字段或者创建新记录的函数，代码如下所示。

```
clear_status(#todo{status=S, who=W} = R) ->
    %% 在此函数内部，S和W绑定了记录里的字段值
    %% values in the record
    %%
    %% R是*整个*记录
    R#todo{status=finished}
    %% ...
```

要匹配某个类型的记录，可以这样编写函数定义。

```
do_something(X) when is_record(X, todo) ->
    %% ...
```

这个子句会在X是todo类型的记录时匹配成功。

5.2.4 记录是元组的另一种形式

记录其实就是元组。

```
9> X2.
#todo{status = done,who = joe,text = "Fix errata in book"}
```

现在我们要让shell忘掉todo的定义。

```
10> rf(todo).
ok
11> X2.
{todo,done,joe,"Fix errata in book"}
```

在第10行里，rf(todo)命令使shell忘了todo记录的定义。因此，现在打印X2时，shell将X2显示成一个元组。其实它们在系统内部都是元组，但记录提供了方便的语法，让你可以用名称而非位置来指明不同的元素。

5.3 映射组：关联式键–值存储

映射组从Erlang的R17版开始可供使用。

映射组具有下列属性。

- 映射组的语法与记录相似，不同之处是省略了记录名，并且键–值分隔符是=>或:=。
- 映射组是键–值对的关联性集合。
- 映射组里的键可以是任何全绑定的Erlang数据类型（即数据结构里没有任何未绑定变量）。
- 映射组里的各个元素根据键进行排序。
- 在不改变键的情况下更新映射组是一种节省空间的操作。
- 查询映射组里某个键的值是一种高效的操作。
- 映射组有着明确的顺序。

我们将在下面几节里更详细地介绍映射组。

5.3.1 映射组语法

映射组的写法依照以下语法：

```
#{ Key1 Op Val1, Key2 Op Val2, ..., KeyN Op ValN }
```

它的语法与记录相似，但是散列符号（即#）之后没有记录名，而Op是=>或:=这两个符号的

其中一个。

键和值可以是任何有效的Erlang数据类型。举个例子，假设要创建一个包含a、b两个键的映射组。

```
1> F1 = #{ a => 1, b => 2 }.
#{ a => 1, b => 2 }.
```

或者假设要创建一个带有非原子键的映射组。

```
2> Facts = #{ {wife,fred} => "Sue", {age, fred} => 45,
    {daughter,fred} => "Mary",
    {likes, jim} => [...]}.
#{ {age, fred} => 45, {daughter,fred} => "Mary", ...}
```

映射组在系统内部是作为有序集合存储的，打印时总是使用各键排序后的顺序，与映射组的创建方式无关。这里有一个例子：

```
3> F2 = #{ b => 2, a => 1 }.
#{ a => 1, b => 2 }.
4> F1 = F2.
#{ a => 1, b => 2 }.
```

要基于现有的映射组更新一个映射组，我们会使用如下语法，其中的Op（更新操作符）是=>或:=：

```
NewMap = OldMap # { K1 Op V1,...,Kn Op Vn }
```

表达式K => V有两种用途，一种是将现有键K的值更新为新值V，另一种是给映射组添加一个全新的K-V对。这个操作总是成功的。

表达式K := V的作用是将现有键K的值更新为新值V。如果被更新的映射组不包含键K，这个操作就会失败。

```
5> F3 = F1#{ c => xx }.
#{ a => xx, b => 2 , c => xx}
6> F4 = F1#{ c := 3}
** exception error: bad argument
key c does not exist in old map
```

使用:=操作符有两个重要原因。首先，如果拼错了新键的名称，我们希望会有错误发生。如果创建了一个映射组Var = #{keypos => 1, ...}，然后用Var #{key_pos := 2 }更新它，那么几乎可以肯定拼错了键名，而我们需要知道这一点。第二个原因和效率有关。如果在映射组更新操作里只使用:=操作符，那么我们就知道新旧映射组都带有一组相同的键，因此可以共享相同的键描述符。假如我们有一个包含数百万映射组的列表，并且它们的各个键都相同，那么所节省的空间是很可观的。

使用映射组的最佳方式是在首次定义某个键时总是使用Key => Val，而在修改具体某个键的值时都使用Key := Val。

5.3.2　模式匹配映射组字段

用来编写映射组的=>语法还可以作为映射组模式使用。和之前一样，映射组模式里的键不能包含任何未绑定变量，但是值现在可以包含未绑定变量了（在模式匹配成功后绑定）。

其他语言里的映射组

请注意，Erlang的映射组与其他许多语言里的相应结构有着非常不同的工作方式。要阐释这一点，我们可以来看看在JavaScript里会发生什么。

假设在JavaScript里做下面这些事：

```
var x = {status:'old', task:'feed cats'};
var y = x;
y.status = 'done';
```

y的值是对象{status:'done', task:'feed cats'}。这没什么疑问。但奇怪的是，x也变成了{status:'done', task:'feed cats'}。这对Erlang程序员而言是个很大的意外。我们的确修改了变量x里某个字段的值，但并不是通过操作x，而是给变量y的某个字段指派了一个值。通过别名指针修改x会导致很多微妙的错误，调试起来会非常困难。

逻辑上等价的Erlang代码如下：

```
D1 = {status=>old, task=>'feed cats'},
D2 = D1#{status := done},
```

在Erlang代码里，变量D1和D2不会改变它们的初始值。D2的表现就像是对D1做了深层复制[①]一样。事实上，深层复制并没有发生，Erlang系统只复制了内部结构里的某些必要部分，以形成创建了复制物的假象。因此，创建一个看似某个对象深层复制物的操作是极其轻量的。

```
1> Henry8 = #{ class => king, born => 1491, died => 1547 }.
#{ born => 1491, class=> king, died => 1547 }.
2> #{ born => B } = Henry8.
#{ born => 1491, class=> king, died => 1547 }.
3> B.
1491
4> #{ D => 1547 }.
* 4: variable 'D' unbound
```

我们在第1行创建了一个包含亨利八世（Henry VIII）信息的新映射组。在第2行里创建了一个模式来从映射组提取关联born键的值。模式匹配成功后，shell打印出整个映射组的值。在第3行打印了变量B的值。

在第4行里我们试图找到一个值为1547的未知键（D）。但是shell打印出一个错误，因为映射组里所有的键都必须是全绑定的数据类型，而D没有定义过。

请注意，映射组模式里键的数量可能少于所匹配映射组里键的数量。

① 深层复制（deep copy）是指数据被真正复制到了新的内存地址，而不是像浅层复制（shallow copy）那样仅仅复制了内存指针。——译者注

可以在函数的头部使用包含模式的映射组，前提是映射组里所有的键都是已知的。举个例子，定义一个count_characters(Str)函数，让它返回一个映射组，内含某个字符串里各个字符的出现次数。

```
count_characters(Str) ->
   count_characters(Str, #{}).

count_characters([H|T], #{ H => N }=X) ->
   count_characters(T, X#{ H := N+1 });
count_characters([H|T], X) ->
   count_characters(T, X#{ H => 1 });
count_characters([], X) ->
   X.
```

下面是一个例子：

```
1> count_characters("hello").
#{101=>1,104=>1,108=>2,111=>1}
```

于是我们知道字符h（ASCII码是101）出现了一次，以此类推。count_characters/2有两个值得注意的地方。子句1里的映射组内变量H定义于映射组之外，因此是绑定的（这是必需的）。在子句2里，我们用map_extend来给映射组添加一个新键。

5.3.3　操作映射组的内置函数

还有许多函数能操作映射组，它们是maps模块的一部分。

- maps:new() -> #{}
  返回一个新的空映射组。

- erlang:is_map(M) -> bool()
  如果M是映射组就返回true，否则返回false。它可以用在关卡测试或函数主体中。

- maps:to_list(M) -> [{K1,V1},..., {Kn,Vn}]
  把映射组M里的所有键和值转换成一个键值列表。键在生成的列表里严格按升序排列。

- maps:from_list([{K1,V1},...,{Kn,Vn}]) -> M
  把一个包含键值对的列表转换成映射组M。如果同样的键不止一次出现，就使用列表里第一个键所关联的值，后续的值都会被忽略。

- maps:map_size(Map) -> NumberOfEntries
  返回映射组里的条目数量。

- maps:is_key(Key, Map) -> bool()
  如果映射组包含一个键为Key的项就返回true，否则返回false。

- maps:get(Key, Map) -> Val

  返回映射组里与Key关联的值，否则抛出一个异常错误。

- maps:find(Key, Map) -> {ok, Value} | error

  返回映射组里与Key关联的值，否则返回error。

- maps:keys(Map) -> [Key1,..KeyN]

  返回映射组所含的键列表，按升序排列。

- maps:remove(Key, M) -> M1

  返回一个新映射组M1，除了键为Key的项（如果有的话）被移除外，其他与M一致。

- maps:without([Key1, ..., KeyN], M) -> M1

  返回一个新映射组M1，它是M的复制，但移除了带有[Key1,...,KeyN]列表里这些键的元素。

- maps:difference(M1, M2) -> M3

  M3是M1的复制，但移除了那些与M2里的元素具有相同键的元素。

它的行为类似下面这种定义：

```
maps:difference(M1, M2) ->
    maps:without(maps:keys(M2), M1).
```

5.3.4　映射组排序

映射组在比较时首先会比大小，然后再按照键的排序比较键和值。

如果A和B是映射组，那么当maps:size(A) < maps:size(B)时A < B。

如果A和B是大小相同的映射组，那么当maps:to_list(A) < maps:to_list(B)时A < B。

举个例子，A = #{age => 23, person => "jim"}小于B = # {email => "sue@somplace.com", name => "sue"}。这是因为A的最小键（age）比B的最小键（email）更小。

当映射组与其他Erlang数据类型相比较时，因为我们认为映射组比列表或元组"更复杂"，所以映射组总是会大于列表或元组。

映射组可以通过io:format里的~p选项输出，并用io:read或file:consult读取。

5.3.5　以JSON为桥梁

熟悉JSON的读者会注意到映射组与JSON数据类型之间的相似性。有两个内置函数可以让映射组和JSON数据相互转换。

- maps:to_json(Map) -> Bin

  把一个映射组转换成二进制型，它包含用JSON表示的该映射组。二进制型会在第7章中展

开讨论。请注意，不是所有的映射组都能转换成JSON数据类型。映射组里所有的值都必须是能用JSON表示的对象。例如，值不能包含的对象有fun、进程标识符和引用等。如果有任何的键或值不能用JSON表示，`maps:to_json`就会失败。

- maps:from_json(Bin) -> Map
  把一个包含JSON数据的二进制型转换成映射组。

- maps:safe_from_json(Bin) -> Map
  把一个包含JSON数据的二进制型转换成映射组。`Bin`里的任何原子必须在调用此内置函数前就已存在，否则就会抛出一个异常错误。这样做是为了防止创建大量的新原子。出于效率的原因，Erlang不会垃圾回收（garbage collect）原子，所以连续不断地添加新原子会（在很长一段时间后）让Erlang虚拟机崩溃。

上面两种定义里的`Map`都必须是`json_map()`类型的实例，它的定义如下（类型的定义会在第9章中介绍）：

```
-type json_map() = [{json_key(), json_value()}].
```

其中：

```
-type json_key() =
    atom() | binary() | io_list()
```

并且：

```
-type json_value() =
    integer() | binary() | float() | atom() | [json_value()] | json_map()
```

JSON对象与Erlang值的映射关系如下。

- JSON的数字用Erlang的整数或浮点数表示。
- JSON的字符串用Erlang的二进制型表示。
- JSON的列表用Erlang的列表表示。
- JSON的true和false用Erlang的原子`true`和`false`表示。
- JSON的对象用Erlang的映射组表示，但是有限制：映射组里的键必须是原子、字符串或二进制型，而值必须可以用JSON的数据类型表示。

当来回转换JSON数据类型时，应当注意一些特定的转换限制。Erlang对整数提供了无限的精度。所以，Erlang会很自然地把映射组里的某个大数转换成JSON数据里的大数，而解码此JSON数据的程序不一定能理解它。

在第18章，你会了解如何结合使用映射组、JSON数据和WebSocket，实现一种与在Web服务器内运行的程序进行通信的简单方法。

到目前为止，我们已经介绍了Erlang里所有创建复合数据结构的方法。我们知道了列表是放置可变数量项目的容器，而元组是放置固定数量项目的容器。记录的作用是给元组里的各个元素添加符号名称，映射组则被当作关联数组使用。

5

下一章将介绍错误处理。在此之后，我们会回到顺序编程上来，然后介绍尚未提及的二进制型和位语法。

5.4　练习

(1) 配置文件可以很方便地用JSON数据表示。请编写一些函数来读取包含JSON数据的配置文件，并将它们转换成Erlang的映射组。再编写一些代码，对配置文件里的数据进行合理性检查。

(2) 编写一个`map_search_pred(Map, Pred)`函数，让它返回映射组里第一个符合条件的`{Key,Value}`元素（条件是`Pred(Key, Value)`为`true`）。

(3) 高级练习：查找Ruby散列类的手册页。制作一个模块，加入这个Ruby类里你认为适合Erlang的方法。

第 6 章　顺序程序的错误处理

Erlang最初被设计用来编写容错式系统，这种系统原则上应该永不停歇。这就意味着运行时的错误处理是至关重要的。在Erlang里，我们会非常严肃地对待错误处理。当错误发生时，需要发现并纠正它，然后继续。

常见的Erlang应用程序是由几十到几百万个并发进程组成的。拥有大量进程改变了我们对错误处理的看法。在只有一个进程的顺序编程语言里，关键的一点是不能让这个进程崩溃。但如果有了大量的进程，单个进程的崩溃就不那么重要了，前提是其他某些进程能察觉这个崩溃，并接手崩溃进程原本应该做的事情。

要构建真正容错的系统，我们需要不止一台计算机：毕竟，崩溃的可能是整台计算机。因此，故障检测和在别处重启计算这个概念必须扩展到联网的计算机上。

要完全理解错误处理，必须先了解顺序程序里的错误处理，理解之后再来看如何处理大量并行进程里的错误。这一章讨论的是前者。处理并发进程里的错误由第13章负责，而构建一组合作纠正错误的进程则是23.5节的主题。

6.1　处理顺序代码里的错误

每当我们在Erlang里调用某个函数后，下面两件事中的一件就会发生：函数或者返回一个值，或者出现了问题。我们在上一章里见过这种例子。还记得`cost`函数吗？

shop.erl

```
cost(oranges)   -> 5;
cost(newspaper) -> 8;
cost(apples)    -> 2;
cost(pears)     -> 9;
cost(milk)      -> 7.
```

运行这个函数会发生以下情况：

```
1> shop:cost(apples).
2
2> shop:cost(socks).
** exception error: no function clause matching
   shop:cost(socks) (shop.erl, line 5)
```

当我们调用`cost(socks)`后，程序崩溃了。发生这种情况的原因是函数定义里没有任何一个子句能匹配调用的参数。

调用`cost(socks)`完全是无稽之谈。这个函数不能返回任何合理的值，因为袜子的价格未被定义。在这种情况下，系统没有返回值，而是抛出了一个异常错误——这是"崩溃"的技术性说法。

我们不会去尝试修复这个错误，因为这是不可能的。我们不知道袜子的价格，因此无法返回一个值。应该由`cost(socks)`的调用者来决定函数崩溃后该怎么处理。

异常错误发生于系统遇到内部错误时，或者通过在代码里显式调用`throw(Exception)`、`exit(Exception)`或`error(Exception)`触发。当我们执行`cost(socks)`时会发生一个模式匹配错误。没有任何一个子句定义了袜子的价格，所以系统会自动生成一个错误。

会触发异常错误的典型内部错误有模式匹配错误（没有一个函数子句能成功匹配），用错误类型的参数调用内置函数（比如用一个整数作为参数调用`atom_to_list`），以及用带有错误值的参数调用内置函数（比如试图让某个数字除以0）。

注意　许多编程语言建议你应该做防御式编程（defensive programming）并检查所有函数的参数。而在Erlang里，防御式编程是内建的。在描述函数的行为时应该只考虑合法的输入参数，其他所有参数都将导致内部错误并自动被检测到。永远不能让函数对非法的参数返回值，而是应该抛出一个异常错误。这条规则被称为"任其崩溃"。

可以通过调用下面的某个内置函数来显式生成一个错误。

- exit(Why)

  当你确实想要终止当前进程时就用它。如果这个异常错误没有被捕捉到，信号`{'EXIT',Pid,Why}`就会被广播给当前进程链接的所有进程。我们还没有遇到过信号，在13.3节里会进行详细讨论。信号和错误消息非常相似，但在这里只是点到为止。

- throw(Why)

  这个函数的作用是抛出一个调用者可能想要捕捉的异常错误。在这种情况下，我们注明了被调用函数可能会抛出这个异常错误。有两种方法可以代替它使用：可以为通常的情形编写代码并且有意忽略异常错误，也可以把调用封装在一个`try...catch`表达式里，然后对错误进行处理。

- error(Why)

  这个函数的作用是指示"崩溃性错误"，也就是调用者没有准备好处理的非常严重的问题。它与系统内部生成的错误差不多。

Erlang有两种方法来捕捉异常错误。第一种是把抛出异常错误的调用函数封装在一个`try...catch`表达式里，另一种是把调用封装在一个`catch`表达式里。

6.2 用 try...catch 捕捉异常错误

如果你熟悉Java，理解`try...catch`表达式就不会有任何问题。Java可以用以下语法捕捉异常错误。

```
try {
    block
} catch (exception type identifier) {
    block
} catch (exception type identifier) {
    block
} ...
finally {
    block
}
```

Erlang有一个非常相似的结构，看起来就像这样：

```
try FuncOrExpressionSeq of
    Pattern1 [when Guard1] -> Expressions1;
    Pattern2 [when Guard2] -> Expressions2;
    ...
catch
    ExceptionType1: ExPattern1 [when ExGuard1] -> ExExpressions1;
    ExceptionType2: ExPattern2 [when ExGuard2] -> ExExpressions2;
    ...
after
    AfterExpressions
end
```

6.2.1 try...catch具有一个值

请记住，Erlang里的一切都是表达式，而表达式都具有值。之前在“`if`表达式”里提到过这一点，当时讨论的是为什么`if`表达式没有`else`部分。这就意味着`try...end`这个表达式也具有一个值。因此，我们可以编写这样的代码：

```
f(...) ->
...
X = try ... end,
Y = g(X),
...
```

更多情况下，并不需要`try...catch`表达式的值。所以只需要这样写：

```
f(...) ->
    ...
    try ... end,
    ...
    ...
```

请注意try...catch表达式和case表达式之间的相似性。

```
case Expression of
    Pattern1 [when Guard1] -> Expressions1;
    Pattern2 [when Guard2] -> Expressions2;
    ...
end
```

try...catch就像是case表达式的强化版。它基本上就是case表达式加上最后的catch和after区块。

try...catch的工作方式如下：首先执行FuncOrExpessionSeq。如果执行过程没有抛出异常错误，那么函数的返回值就会与Pattern1（以及可选的关卡Guard1）、Pattern2等模式进行匹配，直到匹配成功。如果能匹配，那么整个try...catch的值就通过执行匹配模式之后的表达式序列得出。

如果FuncOrExpressionSeq在执行中抛出了异常错误，那么ExPattern1等捕捉模式就会与它进行匹配，找出应该执行哪一段表达式序列。ExceptionType是一个原子（throw、exit和error其中之一），它告诉我们异常错误是如何生成的。如果省略了ExceptionType，就会使用默认值throw。

注意　Erlang运行时系统所检测到的内部错误总是带有error标签。

关键字after之后的代码是用来在FuncOrExpressionSeq结束后执行清理的。这段代码一定会被执行，哪怕有异常错误抛出也是如此。after区块的代码会在try或catch区块里的Expressions代码完成后立即运行。AfterExpressions的返回值会被丢弃。

如果你来自Ruby，所有这一切看起来应该十分熟悉。在Ruby里编写的代码风格与之类似。

```
begin
   ...
rescue
   ...
ensure
   ...
end
```

关键字有所不同，但它们的行为是相似的。

6.2.2　简写法

可以省略try...catch表达式的多个部分。这段代码：

```
try F
catch
   ...
end
```

就等于这一段：

```
try F of
  Val -> Val
catch
   ...
end
```

除此之外，after部分也可以省略。

6.2.3 try...catch编程样例

设计应用程序时，如果某段代码的作用是捕捉错误，那么通常会设法确保它能捕捉到函数可能生成的一切错误。

下面这两个函数对此进行了演示。第一个函数会生成三种不同类型的异常错误，并有两个常规的返回值。

try_test.erl

```
generate_exception(1) -> a;
generate_exception(2) -> throw(a);
generate_exception(3) -> exit(a);
generate_exception(4) -> {'EXIT', a};
generate_exception(5) -> error(a).
```

现在编写一个封装函数，用它在一个try...catch表达式里调用generate_exception。

try_test.erl

```
demo1() ->
     [catcher(I) || I <- [1,2,3,4,5]].

catcher(N) ->
   try generate_exception(N) of
       Val -> {N, normal, Val}
   catch
       throw:X -> {N, caught, thrown, X};
       exit:X  -> {N, caught, exited, X};
       error:X -> {N, caught, error, X}
   end.
```

运行它会带来以下结果：

```
> try_test:demo1().
[{1,normal,a},
 {2,caught,thrown,a},
 {3,caught,exited,a},
 {4,normal,{'EXIT',a}},
 {5,caught,error,a}]
```

它展示了捕捉与区分一个函数所能抛出的所有异常错误形式。

6.3　用 catch 捕捉异常错误

另一种捕捉异常错误的方法是使用基本语法catch。catch语法和try...catch里的catch区块不是一回事（这是因为catch语句早在try...catch被引入之前就已经是Erlang语言的一部分了）。

异常错误如果发生在catch语句里，就会被转换成一个描述此错误的{'EXIT', ...}元组。要演示这一点，可以在一个catch表达式里调用generate_exception。

try_test.erl

```
demo2() ->
    [{I, (catch generate_exception(I))} || I <- [1,2,3,4,5]].
```

运行它会带来以下结果：

```
2> try_test:demo2().
[{1,a},
 {2,a},
 {3,{'EXIT',a}},
 {4,{'EXIT',a}},
 {5,{'EXIT',
        {a,[{try_test,generate_exception,1,
                    [{file,"try_test.erl"},{line,9}]},
        {try_test,'-demo2/0-lc$^0/1-0-',1,
                    [{file,"try_test.erl"},{line,28}]},
        {try_test,'-demo2/0-lc$^0/1-0-',1,
                    [{file,"try_test.erl"},{line,28}]},
        {erl_eval,do_apply,6,[{file,"erl_eval.erl"},{line,576}]},
        {shell,exprs,7,[{file,"shell.erl"},{line,668}]},
        {shell,eval_exprs,7,[{file,"shell.erl"},{line,623}]},
        {shell,eval_loop,3,[{file,"shell.erl"},{line,608}]}]}}}]
```

如果你将此输出与try...catch一节里的相比，就会发现这两种方法提供了不同量级的调试信息。第一种方法概括了信息，第二种则提供了详细的栈跟踪信息。

6.4　针对异常错误的编程样式

处理异常错误不是什么高精尖技术，下面几节包含了一些常见的代码模式，我们可以在自己的程序里借鉴它们。

6.4.1　改进错误消息

内置函数error/1的一种用途是改进错误消息的质量。如果在调用math:sqrt(X)时提供一个负值的参数，就会发生下面的情况：

```
1> math:sqrt(-1).
** exception error: bad argument in an arithmetic expression
```

```
in function  math:sqrt/1
   called as math:sqrt(-1)
```

可以为其编写一个封装函数来改进错误消息。

lib_misc.erl

```
sqrt(X) when X < 0 ->
    error({squareRootNegativeArgument, X});
sqrt(X) ->
    math:sqrt(X).
```

```
2> lib_misc:sqrt(-1).
** exception error: {squareRootNegativeArgument,-1}
   in function  lib_misc:sqrt/1
```

6.4.2 经常返回错误时的代码

如果你的函数并没有什么"通常的情形",那么多半应该返回{ok, Value}或{error, Reason}这类值，但是请记住，这将迫使所有的调用者必须对返回值做点什么。然后必须二选一，一种是这么写：

```
...
case f(X) of
    {ok, Val} ->
        do_some_thing_with(Val);

    {error, Why} ->
        %% ... 处理这个错误 ...
end,
...
```

这样两种返回值都会被处理。另一种是这么写：

```
...
{ok, Val} = f(X),
do_some_thing_with(Val);
...
```

这样，如果f(X)返回{error, ...}就会抛出一个异常错误。

6.4.3 错误可能有但罕见时的代码

这种情况下，通常要编写能处理错误的代码，就像这个例子一样：

```
try my_func(X)
catch
  throw:{thisError, X} -> ...
  throw:{someOtherError, X} -> ...
end
```

同时，检测错误的代码也应该带有匹配的throw，就像下面这样：

```
my_func(X) ->
    case ... of
       ...
       ... ->
              ... throw({thisError, ...})
       ... ->
              ... throw({someOtherError, ...})
```

6.4.4　捕捉一切可能的异常错误

如果想要捕捉一切可能的错误，就可以使用下面的句式（基于_能匹配一切事物这条规则）：

```
try Expr
catch
    _:_ -> ... 处理所有异常错误的代码 ...
end
```

如果在代码里漏写了标签：

```
try Expr
catch
    _ -> ... 处理所有异常错误的代码 ...
    end
```

就不会捕捉到所有的错误，因为在这种情形下系统会假设标签是默认的throw。

6.5　栈跟踪

捕捉到一个异常错误后，可以调用erlang:get_stacktrace()来找到最近的栈跟踪信息。这里有一个例子：

try_test.erl

```
demo3() ->
    try generate_exception(5)
    catch
       error:X ->
            {X, erlang:get_stacktrace()}
    end.
```

```
1> try_test:demo3().
{a,[{try_test,generate_exception,1,[{file,"try_test.erl"},{line,9}]},
    {try_test,demo3,0,[{file,"try_test.erl"},{line,33}]},
    {erl_eval,do_apply,6,[{file,"erl_eval.erl"},{line,576}]},
    {shell,exprs,7,[{file,"shell.erl"},{line,668}]},
    {shell,eval_exprs,7,[{file,"shell.erl"},{line,623}]},
    {shell,eval_loop,3,[{file,"shell.erl"},{line,608}]}]}
```

上面的跟踪信息展示了试图执行try_test:demo3()时发生了什么。它表明程序在generate_exception/1函数中崩溃，而该函数是在try_test.erl文件的第9行里定义的。

栈跟踪还包含了当前（已崩溃的）函数如果执行成功会返回何处的信息。栈跟踪里的各个元组都是{Mod,Func,Arity,Info}这种形式。Mod、Func和Arity指明了某个函数，Info则包含了栈跟踪里这一项的文件名和行号。

因此，try_test:generate_exception/1本应该返回try_test:demo3()，而后者本应该返回erl_eval:do_apply/6，以此类推。如果某个函数在表达式序列中被调用，那么调用位置和函数将要返回的位置几乎是一样的。如果被调用的函数是表达式序列的最后一个函数，那么此函数的调用位置信息不会保留在栈上。Erlang会对这一类代码进行尾调用优化（last-call optimization），因此栈跟踪信息不会记录函数被调用时的位置，只会记录它将要返回的位置。

分析栈跟踪信息能让我们很好地判断出错误发生时程序的执行位置。通常栈跟踪信息的头两条就足以让你找到错误发生的位置了。

现在，我们了解了顺序程序里的错误处理。要牢记的一个重点是任其崩溃。永远不要在函数被错误参数调用时返回一个值，而是要抛出一个异常错误。要假定调用者会修复这个错误。

6.6 抛错要快而明显，也要文明

为错误编写代码时需要考虑两个关键原则。第一，应该在错误发生时立即将它抛出，而且要抛得明显。有些编程语言采用静默出错的原则，尝试修复错误并继续运行，这会导致代码调试起来异常困难。而在Erlang里，当系统内部或程序逻辑检测出错误时，正确的做法是立即崩溃并生成一段有意义的错误消息。立即崩溃是为了不让事情变得更糟。错误消息应当被写入永久性的错误日志，而且要包含足够多的细节，以便过后查明是哪里出了错。

第二，文明抛错的意思是只有程序员才应该能看到程序崩溃时产生的详细错误消息。程序的用户绝对不能看到这些消息。另一方面，用户应当得到警告，让他们知道有错误发生这一情况，以及可以采取什么措施来弥补错误。

错误消息对程序员来说就像是来之不易的砂金。绝不能任由它们随着屏幕滚动而永远消失。它们应当被保存到一个永久性的日志文件里，以供日后阅读。

到目前为止，我们只涉及了顺序程序里的错误。第13章将介绍如何管理并发程序里的错误，而在23.2节里，我们将了解如何永久性记录错误日志，以使它们不会丢失。

下一章将介绍二进制型和位语法。位语法是Erlang所独有的，它将模式匹配扩展到了位字段（bit field）上，让编写操作二进制数据的程序变得更简单。

6.7 练习

(1) file:read_file(File)会返回{ok, Bin}或者{error, Why}，其中File是文件名，Bin则包含了文件的内容。请编写一个myfile:read(File)函数，当文件可读取时返回Bin，否则抛出一个异常错误。

(2) 重写try_test.erl里的代码，让它生成两条错误消息：一条文明的消息给用户，另一条详细的消息给开发者。

第 7 章

二进制型与位语法

二进制型（binary）是一种数据结构，它被设计成用一种节省空间的方式来保存大批量的原始数据。Erlang虚拟机对二进制型的输入、输出和消息传递都做了优化，十分高效。

如果要保存大批量的无结构数据内容，二进制型应当是首选，比如大型字符串或文件的内容。

在大多数情况下，二进制型里的位数都会是8的整数倍，因此对应一个字节串。如果位数不是8的整数倍，就称这段数据为位串（bitstring）。所以当我们说位串时，是在强调数据里的位数不是8的整数倍。

把二进制型、位串和位级模式匹配引入Erlang是为了简化网络编程，因为我们通常希望深入探索协议包里的位级和字节级结构。

在这一章里，首先将详细介绍二进制型。二进制型上的大多数操作同样适用于位串，因此在理解二进制型之后，会重点介绍位串与二进制型的不同之处。

7.1 二进制型

二进制型的编写和打印形式是双小于号与双大于号之间的一列整数或字符串。这里有一个例子：

```
1> <<5,10,20>>.
<<5,10,20>>
2> <<"hello">>.
<<"hello">>
3> <<65,66,67>>
<<"ABC">>
```

在二进制型里使用整数时，它们必须属于0至255这个范围。二进制型`<<"cat">>`是`<<99,97,116>>`的简写形式，也就是说，这个二进制型是由字符串里这些字符的ASCII编码组成的。

和字符串类似，如果某个二进制型的内容是可打印的字符串，shell就会将这个二进制型打印成字符串，否则就打印成一列整数。

可以用内置函数来构建二进制型或提取它里面的元素，也可以使用位语法（参见7.2节）。在这一节里，我们将只关注操作二进制型的内置函数。

操作二进制型

可以用内置函数来操作二进制型，也可以使用binary模块里的函数。binary里导出的许多函数都是以本地代码的形式实现的。以下是其中最重要的一些。

- list_to_binary(L) -> B

  list_to_binary返回一个二进制型，它是通过把io列表（iolist）L里的所有元素压扁后形成的（压扁的意思是移除列表里所有的括号）。io列表本身是循环定义的，它是指一个列表所包含的元素是0..255的整数、二进制型或者其他io列表。

```
1> Bin1 = <<1,2,3>>.
<<1,2,3>>
2> Bin2 = <<4,5>>.
<<4,5>>
3> Bin3 = <<6>>.
<<6>>
4> list_to_binary([Bin1,1,[2,3,Bin2],4|Bin3]).
<<1,2,3,1,2,3,4,5,4,6>>
```

注意 第1行等号两边的空白是必需的。如果没有空白，Erlang的分词器就会把第二个符号看作是原子=<，即小于等于操作符。有时候必须在二进制型数据的周围加上空白或括号来避免语法错误。

- split_binary(Bin, Pos) -> {Bin1, Bin2}

  这个函数在Pos处把二进制型Bin一分为二。

```
1> split_binary(<<1,2,3,4,5,6,7,8,9,10>>, 3).
{<<1,2,3>>,<<4,5,6,7,8,9,10>>}
```

- term_to_binary(Term) -> Bin

  这个函数能把任何Erlang数据类型转换成一个二进制型。

  term_to_binary生成的二进制型使用了所谓的外部数据格式（external term format）。数据类型通过term_to_binary转换成二进制型后可以被保存在文件里，作为消息通过网络发送，等等，而转换前的初始数据类型可以在稍后重建。对于在文件里保存复杂数据结构，或者向远程机器发送复杂数据结构而言，这是极其有用的。

- binary_to_term(Bin) -> Term

  这是term_to_binary的逆向函数。

```
1> B = term_to_binary({binaries,"are", useful}).
<<131,104,3,100,0,8,98,105,110,97,114,105,101,115,107,
0,3,97,114,101,100,0,6,117,115,101,102,117,108>>
2> binary_to_term(B).
{binaries,"are",useful}
```

- byte_size(Bin) -> Size

  这个函数返回二进制型里的字节数。

```
1> byte_size(<<1,2,3,4,5>>).
5
```

在所有这些函数里，term_to_binary和binary_to_term无疑是我的最爱。它们非常有用。term_to_binary可以把任何数据类型转换成一个二进制型。在这个二进制型的内部（如果窥视一下），你会发现用“Erlang外部数据格式”（在Erlang文档里有定义[①]）保存的数据。一旦把某个数据类型转换成二进制型，就可以把它作为消息通过套接字发送出去，或者保存在文件里。这是实现分布式Erlang系统的基本方法，也在许多数据库内部使用。

7.2　位语法

位语法是一种表示法，用于从二进制数据里提取或加入单独的位或者位串。当你编写底层代码，以位为单位打包和解包二进制数据时，就会发现位语法是极其有用的。开发位语法是为了进行协议编程（这是Erlang的强项），以及生成操作二进制数据的高效代码。

假设要把三个变量（X、Y和Z）打包进一个16位的内存区域。X应当在结果里占据3位，Y应当占据7位，而Z应当占据6位。在大多数语言里这意味着要进行一些麻烦的底层操作，包括位移位（bit shifting）和位掩码（bit masking）。而在Erlang里，只需要这么写：

```
M = <<X:3, Y:7, Z:6>>
```

这段代码会创建一个二进制型并把它保存在变量M里。请注意：M的类型是binary，因为数据的总长度是16位，可以被8整除。如果换一种写法，把X的大小改成2位：

```
M = <<X:2, Y:7, Z:6>>
```

那么M的总位数是15，因此最终数据结构的类型是bitstring。

完整的位语法要略微复杂一些，所以我们会一步步加以介绍。首先，通过一些简单的代码来看看如何打包和解包16位字长的RGB颜色数据。然后，深入位语法表达式的细节。最后，展示三个使用位语法的代码示例，它们来源于真实的程序。

7.2.1　打包和解包 16 位颜色

我们将从一个非常简单的例子开始。假设想要表示一种16位RGB颜色。我们决定给红色通道分配5位，绿色通道分配6位，剩下的5位则分配给蓝色通道。（让绿色通道多使用1位是因为人眼对绿光更敏感。）

可以创建一个16位的内存区域Mem，让它包含单组RGB值，就像下面这样：

---

① http://erlang.org/doc/apps/erts/erl_ext_dist.html

```
1> Red = 2.
2
2> Green = 61.
61
3> Blue = 20.
20
4> Mem = <<Red:5, Green:6, Blue:5>>.
<<23,180>>
```

请注意，我们在表达式4里创建了一个包含16位数据的双字节二进制型。shell将它打印为`<<23,180>>`。

要打包这段内存，只需写下表达式`<<Red:5, Green:6, Blue:5>>`。

要把这个二进制型解包成整数变量`R1`、`G1`和`B1`，可以编写一个模式。

```
5> <<R1:5, G1:6, B1:5>> = Mem.
<<23,180>>
6> R1.
2
7> G1.
61
8> B1.
20
```

这真的很简单。不妨试着用你最喜欢的编程语言里的位移位和逻辑and/or来实现它。

实际上，用位语法能做的比这个简单例子所展示的要多得多，但首先需要掌握一套相当复杂的语法。一旦做到了，就能编写特别简短的代码来打包和解包复杂的二进制数据结构了。

7.2.2 位语法表达式

位语法表达式被用来构建二进制型或位串。它们的形式如下：

```
<<>>
<<E1, E2, ..., En>>
```

每个`Ei`元素都标识出二进制型或位串里的一个*片段*。单个`Ei`元素可以有4种形式。

```
Ei = Value |
     Value:Size |
     Value/TypeSpecifierList |
     Value:Size/TypeSpecifierList
```

如果表达式的总位数是8的整数倍，就会构建一个二进制型，否则构建一个位串。

当你构建二进制型时，`Value`必须是已绑定变量、字符串，或是能得出整数、浮点数或二进制型的表达式。当它被用于模式匹配操作时，`Value`可以是绑定或未绑定的变量、整数、字符串、浮点数或二进制型。

`Size`必须是一个能够得出整数的表达式。在模式匹配里，`Size`必须是一个整数，或者是值为整数的已绑定变量。如果`Size`在模式里所处的位置需要有一个值，它就必须是已绑定变量。二

进制型里某个Size的值可以通过之前的模式匹配获得。例如，下面的模式：

```
<<Size:4, Data:Size/binary, ...>>
```

是个合法的模式，因为Size的值已经由二进制型的前四位解包获得，然后被用于指明二进制型内下一个片段的大小。

Size的值指明了片段的大小。它的默认值取决于不同的数据类型，对整数来说是8，浮点数则是64，如果是二进制型就是该二进制型的大小。在模式匹配里，默认值只对最后那个元素有效。如果未指定片段的大小，就会采用默认值。

TypeSpecifierList（类型指定列表）是一个用连字符分隔的列表，形式为End-Sign-Type-Unit。前面这些项中的任何一个都可以被省略，各个项也可以按任意顺序排列。如果省略了某一项，系统就会使用它的默认值。

类型指定列表里的各项可以有如下这些值。

- End可以是big | little | native

  它指定机器的字节顺序。native是指在运行时根据机器的CPU来确定。默认值是big，也就是网络字节顺序（network byte order）。这一项只和从二进制型里打包和解包整数与浮点数有关。当你在不同字节顺序的机器上打包和解包二进制型里的整数时，应当注意设置正确的字节顺序。

  编写位语法表达式时，可能有必要先做些试验。为了确定自己没有弄错，你可以尝试下面的shell命令：

  ```
  1> {<<16#12345678:32/big>>,<<16#12345678:32/little>>,
  <<16#12345678:32/native>>,<<16#12345678:32>>}.
  {<<18,52,86,120>>,<<120,86,52,18>>,
  <<120,86,52,18>>,<<18,52,86,120>>}
  ```

  输出结果展示了位语法是如何把整数打包进二进制型的。

  如果你还是不放心，term_to_binary和binary_to_term可以帮你搞定打包和解包整数的工作。因此，你可以在高位优先（big-endian）的机器上创建一个包含整数的元组，然后用term_to_binary把它转换成二进制型并发送至低位优先（little-endian）的机器。最后，在低位优先的机器上运行binary_to_term，这样元组里所有整数的值都会是正确的。

- Sign可以是signed|unsigned

  这个参数只用于模式匹配。默认值是unsigned。

- Type可以是integer|float|binary|bytes|bitstring|bits|utf8|utf16|utf32

  默认值是integer。

- Unit的写法是unit:1|2|…256

  integer、float和bitstring的Unit默认值是1，binary则是8。utf8、utf16和utf32类型无需提供值。

一个片段的总长度是`Size x Unit`字节。`binary`类型的片段长度必须是8的整数倍。

如果你觉得位语法的描述有点令人生畏，别着急。正确设置位语法的模式可能会很难。最好的办法是在shell里试验你需要的模式直到能用为止，然后再把结果复制粘贴到你的程序里。我就是这么做的。

7.2.3 位语法的真实例子

学习位语法需要一点额外的努力，但是它带来的好处是巨大的。这一节里有三个来源于现实生活的例子。所有这些代码都是从真实的程序中剪切复制而来的。

第一个例子是寻找MPEG音频数据里的同步点。这个例子展示了位语法模式匹配的威力。它的代码非常容易理解，与MPEG头帧规范的对应关系很清晰。第二个例子是构建微软通用对象文件格式（Microsoft Common Object File Format，简称COFF）的二进制数据文件。打包和解包二进制数据文件（例如COFF）的典型做法是使用二进制型和二进制模式匹配。最后的例子展示了如何解包一个IPv4数据报（datagram）。

1. 寻找MPEG数据里的同步帧

假设要编写一个操作MPEG音频数据的程序，比如一个用Erlang编写的流媒体服务器，或者提取描述某个MPEG音频流内容的数据标签。要做到这一点，需要识别并同步某个MPEG流的数据帧。

7

MPEG音频数据是由许多的帧组成的。每一帧都有它自己的帧头，后面跟着音频信息。由于没有文件头，所以原则上可以把一个MPEG文件切成多个片段，而每一段都可以独立播放。任何读取MPEG流的软件都应当能找到头帧，然后同步MPEG数据。

MPEG头以一段11位的帧同步（frame sync）信息开头，它包含11个连续的1位，后面跟着描述后续数据的信息：

```
AAAAAAAA AAABBCCD EEEEFFGH IIJJKLMM
```

AAAAAAAAAAA	同步字（11位，全部为1）
BB	2位的MPEG音频版本ID
CC	2位的层描述
D	这1位是保护位
……	……

这些位的准确含义不是我们的关注点。基本上，只要知道了从`A`到`M`的值，就能计算出一个MPEG帧的总长度。

为了找到同步点，首先假设我们成功定位到了某个MPEG头的起始位置，然后试着计算该帧的长度。之后可能会发生以下某一种情况。

- ❑ 假设正确，因此当向前跳过帧长后，会找到另一个MPEG头。
- ❑ 假设错误。可能没有定位到标记MPEG头起始位置的位序列（由11个连续的1组成）上，

或者因为字的格式不正确而无法计算帧长。

- 假设错误，但定位到的若干字节音乐数据碰巧和MPEG头的起始位置很相似。在这种情况下，可以计算帧长，但是当向前跳过这段距离后，我们无法找到新的MPEG头。

为了确保万无一失，我们将寻找三个连续的MPEG头。同步的程序如下所示：

mp3_sync.erl

```
find_sync(Bin, N) ->
    case is_header(N, Bin) of
        {ok, Len1, _} ->
            case is_header(N + Len1, Bin) of
                {ok, Len2, _} ->
                    case is_header(N + Len1 + Len2, Bin) of
                        {ok, _, _} ->
                            {ok, N};
                        error ->
                            find_sync(Bin, N+1)
                    end;
                error ->
                    find_sync(Bin, N+1)
            end;
        error ->
            find_sync(Bin, N+1)
    end.
```

find_sync尝试寻找三个连续的MPEG头帧。如果Bin的第N个字节是某个头帧的开头，is_header(N, Bin)就会返回{ok, Length, Info}。如果is_header返回了error，那么N指向的就不是正确的帧头。

可以在shell里做个快速测试来确保这段代码能用。

```
1> c(mp3_sync).
{ok, mp3_sync}
2> {ok, Bin} = file:read_file("/home/joe/music/mymusic.mp3").
{ok,<<73,68,51,3,0,0,0,0,33,22,84,73,84,50,0,0,0,28, ...>>
3> mp3_sync:find_sync(Bin, 1).
{ok,4256}
```

这里使用了file:read_file来把整个文件读取到一个二进制型中（参见16.2.4节）。现在来写is_header：

mp3_sync.erl

```
is_header(N, Bin) ->
    unpack_header(get_word(N, Bin)).

get_word(N, Bin) ->
    {_,<<C:4/binary,_/binary>>} = split_binary(Bin, N),
    C.
```

```
unpack_header(X) ->
    try decode_header(X)
    catch
        _:_ -> error
    end.
```

这段代码稍微复杂一些。首先，提取32位数据进行分析（由get_word实现），然后用decode_header解包帧头。现在编写代码，让decode_header在参数不是帧头的开头时崩溃（通过调用exit/1实现）。为了捕捉任何可能的错误，在一个try...catch语句里对decode_header进行调用（更多信息请参阅6.1节）。它还会捕捉到任何由framelength/4里的错误代码所导致的潜在错误。decode_header是最精彩的部分。

mp3_sync.erl

```
decode_header(<<2#11111111111:11,B:2,C:2,_D:1,E:4,F:2,G:1,Bits:9>>) ->
    Vsn = case B of
              0 -> {2,5};
              1 -> exit(badVsn);
              2 -> 2;
              3 -> 1
          end,
    Layer = case C of
                0 -> exit(badLayer);
                1 -> 3;
                2 -> 2;
                3 -> 1
            end,
    %% Protection = D,
    BitRate = bitrate(Vsn, Layer, E) * 1000,
    SampleRate = samplerate(Vsn, F),
    Padding = G,
    FrameLength = framelength(Layer, BitRate, SampleRate, Padding),
    if
        FrameLength < 21 ->
            exit(frameSize);
        true ->
            {ok, FrameLength, {Layer,BitRate,SampleRate,Vsn,Bits}}
    end;
decode_header(_) ->
    exit(badHeader).
```

秘诀就在于代码第一行的这个神奇表达式。

```
decode_header(<<2#11111111111:11,B:2,C:2,_D:1,E:4,F:2,G:1,Bits:9>>) ->
```

2#11111111111是个二进制整数，因此这个模式匹配11个连续的1位，指派给B2位，指派给C2位，以此类推。请注意，这段代码严格遵循之前给出的位级MPEG头规范。很难再写出比这更漂亮和直接的代码了。这段代码不但漂亮，而且高效。Erlang编译器会把位语法模式转变成高度优化的代码，用最佳方式提取里面的字段。

2. 解包COFF数据

几年前我决定编写一个程序来制作能够在Windows中独立运行的Erlang程序，也就是在任何可运行Erlang的机器上都能生成Windows可执行文件。要做到这一点就需要理解和操作采用微软通用对象文件格式（COFF）的文件。查找COFF的细节信息非常棘手，好在有许多C++程序的API文档。C++程序使用的类型声明有DWORD、LONG、WORD和BYTE，这些类型声明对于那些进行过Windows内部编程的程序员来说并不陌生。

文档对相关的数据结构做了说明，但它是从C或C++程序员的视角编写的。下面是一个典型的C类型定义：

```
typedef struct _IMAGE_RESOURCE_DIRECTORY {
   DWORD Characteristics;
   DWORD TimeDateStamp;
   WORD  MajorVersion;
   WORD  MinorVersion;
   WORD  NumberOfNamedEntries;
   WORD  NumberOfIdEntries;
   } IMAGE_RESOURCE_DIRECTORY, *PIMAGE_RESOURCE_DIRECTORY;
```

为了编写这个Erlang程序，我首先定义了4个宏，它们必须包括在Erlang源代码文件内。

```
-define(DWORD, 32/unsigned-little-integer).
-define(LONG,  32/unsigned-little-integer).
-define(WORD,  16/unsigned-little-integer).
-define(BYTE,  8/unsigned-little-integer).
```

注意　宏（macro）会在8.17节中进行解释。要展开这些宏，需要使用?DWORD和?LONG之类的语法。举个例子，宏?DWORD会展开成文本字面量32/unsigned-little-integer。

我有意让这些宏与C里的对应类型具有相同的名称。配备了这些宏之后，我就可以轻松编写一些代码，把图像资源数据解包到二进制型里了。

```
unpack_image_resource_directory(Dir) ->
<<Characteristics      : ?DWORD,
  TimeDateStamp        : ?DWORD,
  MajorVersion         : ?WORD,
  MinorVersion         : ?WORD,
  NumberOfNamedEntries : ?WORD,
  NumberOfIdEntries    : ?WORD, _/binary>> = Dir,
  ...
```

如果对比C和Erlang的代码，你会发现它们非常相似。因此，通过精心选择宏的名称和Erlang代码的布局，就能最大程度地缩小C代码和Erlang代码之间的语义鸿沟，这让我们的程序更容易理解，出错的可能性也会更小。

下一步是解包Characteristics里的数据，以此类推。

Characteristics是一个32位的字，由若干标记组成。用位语法解包它们极其简单，只需

要编写如下代码：

```
<<ImageFileRelocsStripped:1, ImageFileExecutableImage:1, ...>> =
<<Characteristics:32>>
```

代码`<<Characteristics:32>>`把`Characteristics`这个整数转换成32位的二进制型。接下来的代码再把所需的位解包到`ImageFileRelocsStripped`和`ImageFileExecutableImage`这些变量里：

```
<<ImageFileRelocsStripped:1, ImageFileExecutableImage:1, ...>> = ...
```

我让名称与Windows API里的保持一致，以最大程度缩小规范和Erlang程序之间的语义鸿沟。

使用这些宏让解包COFF格式的数据变得——唔，用简单一词恐怕不太恰当，但是这段代码还是比较容易理解的。

3. 解包IPv4数据报头

这个例子演示了用单次模式匹配操作解析互联网协议第4版（Internet Protocol version 4，简称IPv4）的数据报。

```
-define(IP_VERSION, 4).
-define(IP_MIN_HDR_LEN, 5).

...
DgramSize = byte_size(Dgram),
case Dgram of
  <<?IP_VERSION:4, HLen:4, SrvcType:8, TotLen:16,
    ID:16, Flags:3, FragOff:13,
    TTL:8, Proto:8, HdrChkSum:16,
    SrcIP:32,
    DestIP:32, RestDgram/binary>> when HLen >= 5, 4*HLen =< DgramSize ->
        OptsLen = 4*(HLen - ?IP_MIN_HDR_LEN),
        <<Opts:OptsLen/binary,Data/binary>> = RestDgram,
        ...
```

这段代码用一个单独的模式匹配表达式匹配了IP数据报。这个复杂的模式演示了如何简单地提取出不在字节边界内的数据（比如，`Flags`和`FragOff`字段的长度分别是3位和13位）。模式匹配了IP数据报之后，下一次模式匹配操作会提取出此数据报的报头和数据部分。

我们介绍了如何在二进制型上执行位字段操作。再次提醒一下，二进制型的长度必须是8位的整数倍。下一节的主题是位串，它可以用来保存位的序列。

7.3　位串：处理位级数据

对位串（bitstring）的模式匹配是位级操作，这样我们就能在单次操作里打包和解包位的序列。这对编写需要操作位级数据的代码来说极其有用，例如没有按照8位边界对齐的数据或者可变长度数据，它们的数据长度用位而不是字节来表示。

可以在shell里演示位级处理。

```
1> B1 = <<1:8>>.
<<1>>
2> byte_size(B1).
1
3> is_binary(B1).
true
4> is_bitstring(B1).
true
5> B2 = <<1:17>>.
<<0,0,1:1>>
6> is_binary(B2).
false
7> is_bitstring(B2).
true
8> byte_size(B2).
3
9> bit_size(B2).
17
```

位级存储

在大多数编程语言里，最小可寻址存储单元的宽度通常是8位。举个例子，大多数C编译器定义一个char（最小可寻址存储单元）的宽度为8位。操作char里的位十分复杂，因为要单独访问这些位，它们必须被掩码并移位到寄存器里。编写这类代码有些困难而且易于出错。

在Erlang里，最小可寻址的存储单元是1位，位串里各个独立的位序列可以直接访问，无需任何移位和掩码操作。

在前面的例子里，B1是一个二进制型，而B2是一个位串，因为它的长度是17位。我们用语法`<<1:17>>`构建了B2，它被打印成`<<0,0,1:1>>`，也就是说，作为一个二进制型字面量，它的第三个片段是一个长度为1的位串。B2的位大小是17，而字节大小是3（这是包含该位串的二进制型的实际大小）。

使用位串不是件容易的事。举个例子，我们不能把一个位串写入文件或套接字（二进制型则可以），因为文件和套接字使用的单位是字节。

下面用一个例子来结束本节，它会提取出字节里各个单独的位。为了做到这一点，我们将使用一种新的结构，它被称为位推导（bit comprehension）。位推导和二进制型的关系就像列表推导和列表的关系一样。列表推导遍历列表并返回列表。位推导遍历二进制型并生成列表或二进制型。

这个例子展示了如何提取出字节里的各个位。

```
1> B = <<16#5f>>.
<<"_">>
2> [ X || <<X:1>> <= B].
[0,1,0,1,1,1,1,1]
3> << <<X>> || <<X:1>> <= B >>.
<<0,1,0,1,1,1,1,1>>
```

第1行制作了一个包含单个字节的二进制型。`16#5f`是一个十六进制的常量。shell把它打印成了`<<"_">>`，因为`16#5f`是_字符的ASCII编码。第2行里的语法`<<<X:1>>`是一个模式，代表1位。它的结果是一个包含字节里各个位的列表。第3行和第2行类似，唯一的区别是我们构建了一个包含这些位的二进制型，而不是列表。

位推导的语法不会在这里介绍，你可以在Erlang参考手册[1]里找到。更多有关位串处理的示例可以在论文Bit-Level Binaries and Generalized Comprehensions in Erlang[2]里找到。

现在我们已经了解了二进制型和位串。每当我们想要管理大量的无结构数据时，Erlang系统就会在内部使用二进制型。在后面的几章里，我们将看到如何把二进制型作为消息通过套接字发送出去，以及如何保存在文件里。

对于顺序编程，我们已经介绍的差不多了。剩下的都是一些零散的主题，没有什么特别重要或令人激动的，但是了解它们还是很有用的。

7.4 练习

(1) 编写一个函数来反转某个二进制型里的字节顺序。

(2) 编写一个`term_to_packet(Term) -> Packet`函数，通过调用`term_to_binary(Term)`来生成并返回一个二进制型，它内含长度为4个字节的包头N，后跟N个字节的数据。

(3) 编写一个反转函数`packet_to_term(Packet) -> Term`，使它成为前一个函数的逆向函数。

(4) 按照4.1.3节的样式编写一些测试，测一下之前的两个函数是否能正确地把数据类型编码成数据包（packet），以及通过解码数据包来复原最初的数据类型。

(5) 编写一个函数来反转某个二进制型所包含的位。

7

① http://www.erlang.org/doc/reference_manual/users_guide.html

② http://user.it.uu.se/~pergu/papers/erlang05.pdf

第 8 章

Erlang顺序编程补遗

Erlang顺序编程的剩余部分是一些零零散散的东西，你必须了解它们，而它们无法归类于其他主题。这些主题没有什么特定的顺序，为了便于参考，我们将依照英文字母顺序来对它们进行介绍。涉及的主题如下。

- `apply`

  它通过函数名和参数计算该函数的值，其中的函数名和模块名是动态计算得出的。
- 算术表达式

  这里定义了所有合法的算术表达式。
- 元数

  一个函数的元数是这个函数所接受的参数数量。
- 属性

  这一节涉及Erlang模块属性的语法和解释。
- 块表达式

  它们是使用`begin`和`end`的表达式。
- 布尔值

  它们是用原子`true`或`false`表示的事物。
- 布尔表达式

  这一节介绍了所有的布尔表达式。
- 字符集

  这里介绍了Erlang使用的字符集。
- 注释

  这一节介绍了注释的语法。
- 动态代码载入

  这一节介绍了动态代码载入的工作原理。
- Erlang的预处理器

  这一节介绍了Erlang在编译前发生了什么。
- 转义序列

  这一节介绍了转义序列的语法，它被用于字符串和原子。

- 表达式和表达式序列

这一节介绍了表达式和表达式序列是如何定义的。

- 函数引用

这一节介绍了如何引用函数。

- 包含文件

这一节介绍了如何在编译时包含文件。

- 列表添加和移除操作符

它们是++和--。

- 宏

这一节介绍了Erlang的宏处理器。

- 模式的匹配操作符

这一节介绍了如何将匹配操作符=用在模式中。

- 数字

这一节介绍了数字的语法。

- 操作符优先级

这一节介绍了所有Erlang操作符的优先级和结合性。

- 进程字典

每个Erlang进程都有一个本地的破坏性存储区域，有时候会很有用。

- 引用

引用是独一无二的符号。

- 短路布尔表达式

这些是不完全求值的布尔表达式。

- 比较数据类型

这一节介绍了所有的数据类型比较操作符，以及各种数据类型的语法顺序。

- 元组模块

这是一种创建“有状态”模块的方法。

- 下划线变量

这些变量会被编译器特殊对待。

建议你快速浏览这些主题，不用仔细阅读它们。只需有个大概的印象，供以后参考即可。

8.1 apply

内置函数apply(Mod, Func, [Arg1, Arg2, ..., ArgN])会将模块Mod里的Func函数应用到Arg1, Arg2, ... ArgN这些参数上。它等价于以下调用：

```
Mod:Func(Arg1, Arg2, ..., ArgN)
```

apply让你能调用某个模块里的某个函数，向它传递参数。它与直接调用函数的区别在于模

块名和/或函数名可以是动态计算得出的。

所有的Erlang内置函数也可以通过apply进行调用，方法是假定它们都属于erlang模块。因此，要构建一个对内置函数的动态调用，可以编写以下代码：

```
1> apply(erlang, atom_to_list, [hello]).
"hello"
```

警告 应当尽量避免使用apply。当函数的参数数量能预先知道时，M:F(Arg1, Arg2, ... ArgN)这种调用形式要比apply好得多。如果使用apply对函数进行调用，许多分析工具就无法得知发生了什么，一些特定的编译器优化也不能进行。所以，尽量少用apply，除非绝对有必要。

apply的Mod参数不必非得是一个原子，也可以是一个元组。如果我们这么调用：

```
{Mod, P1, P2, ..., Pn}:Func(A1, A2, ..., An)
```

那么实际上调用的是下面这个函数：

```
Mod:Func(A1, A2, ..., An, {Mod, P1, P2, ..., Pn})
```

这个技巧会在24.3节中深入讨论。

8.2　算术表达式

下面的表格展示了所有可用的算术表达式。每种算术操作都有1或2个参数，这些参数在表格里显示为“整数”或“数字”（数字的意思是此参数可以是整数或浮点数）。

表3　算术表达式

操作符	描　述	参数类型	优先级
+ X	+ X	数字	1
- X	- X	数字	1
X * Y	X * Y	数字	2
X / Y	X / Y（浮点除法）	数字	2
bnot X	对X执行按位取反（bitwise not）	整数	2
X div Y	X被Y整除	整数	2
X rem Y	X除以Y的整数余数	整数	2
X band Y	对X和Y执行按位与（bitwise and）	整数	2
X + Y	X + Y	数字	3
X - Y	X - Y	数字	3
X bor Y	对X和Y执行按位或（bitwise or）	整数	3
X bxor Y	对X和Y执行按位异或（bitwise xor）	整数	3
X bsl N	把X向左算术位移（arithmetic bitshift）N位	整数	3
X bsr N	把X向右算术位移N位	整数	3

这些操作符相互之间根据优先级结合。一个复杂算术表达式的求值顺序由所含操作符的优先级而定：所有优先级为1的操作符会首先求值，然后轮到所有优先级为2的操作符，以此类推。

可以用括号来改变默认的求值顺序：括号内的表达式会首先求值。优先级相同的操作符遵循向左结合的规则，从左往右分别求值。

8.3 元数

一个函数的元数（arity）是该函数所拥有的参数数量。在Erlang里，同一模块里的两个名称相同、元数不同的函数是完全不同的函数。除了碰巧使用同一个名称外，它们之间毫不相关。

根据惯例，Erlang程序员经常将名称相同、元数不同的函数作为辅助函数使用。这里有一个例子：

lib_misc.erl

```
sum(L) -> sum(L, 0).

sum([], N)    -> N;
sum([H|T], N) -> sum(T, H+N).
```

你看到的是两个不同的函数，一个元数是1，另一个元数是2。

`sum(L)`函数累加列表L里的所有元素。它用到一个名为`sum/2`的辅助函数，但也可以是其他任何名称。即便把辅助函数命名为`hedgehog/2`（刺猬），程序的意思也不会有任何变化。不过，`sum/2`是更好的命名选择，因为它提示程序的读者这是什么，而且还不必发明一个新名称（这总是很困难的）。

我们经常会通过不导出辅助函数来“隐藏”它们。所以，定义`sum(L)`的模块只会导出`sum/1`，而不会导出`sum/2`。

8.4 属性

模块属性的语法是`-AtomTag(...)`，它们被用来定义文件的某些属性。（注意：`-record(...)`和`-include(...)`有着类似的语法，但是不算模块属性。）模块属性有两种类型：预定义型和用户定义型。

8.4.1 预定义的模块属性

下列模块属性有着预先定义的含义，必须放置在任何函数定义之前。

- -module(modname).
  这是模块声明。`modname`必须是一个原子。此属性必须是文件里的第一个属性。按照惯例，`modname`的代码应当保存在名为`modname.erl`的文件里。如果不这么做，自动代码加载就不能正常工作。更多细节请参阅8.10节。

● -import(Mod, [Name1/Arity1, Name2/Arity2,...]).

import声明列举了哪些函数需要导入到模块中。上面这个声明的意思是要从Mod模块导入参数为Arity1的Name1函数，参数为Arity2的Name2函数，等等。

一旦从别的模块里导入了某个函数，调用它的时候就无需指定模块名了。这里有一个例子：

```
-module(abc).
-import(lists, [map/2]).

f(L) ->
    L1 = map(fun(X) -> 2*X end, L),
    lists:sum(L1).
```

调用map时不需要限定模块名，而调用sum时需要把模块名包括在函数调用内。

● -export([Name1/Arity1, Name2/Arity2, ...]).

导出当前模块里的Name1/Arity1和Name2/Arity2等函数。函数只有被导出后才能在模块之外调用。这里有一个例子：

abc.erl

```
-module(abc).
-export([a/2, b/1]).

a(X, Y) -> c(X) + a(Y).
a(X) -> 2 * X.
b(X) -> X * X.
c(X) -> 3 * X.
```

其中导出声明的意思是只有a/2和b/1能在abc模块之外调用。因此，如果在shell里（也就是模块外部）调用abc:a(5)，就会导致一个错误，因为a/1没有从此模块导出。

```
1> abc:a(1,2).
7
2> abc:b(12).
144
3> abc:a(5).
** exception error: undefined function abc:a/1
```

这段错误消息也许会让人困惑。调用abc:a(5)失败的原因是相关函数未定义，而事实上它在模块里有定义，只是没有被导出。

● -compile(Options).

添加Options到编译器选项列表中。Options可以是单个编译器选项，也可以是一个编译器选项列表（选项的相关介绍可以在compile模块的手册页里找到）。

> **注意**　-compile(export_all).这个编译器选项经常会在调试程序时用到。它会导出模块里的所有函数，无需再显式使用-export标识了。

- -vsn(Version).
  指定模块的版本号。Version可以是任何字面数据类型。Version的值没有什么特别的语法或含义，但可以用于分析程序或者作为说明文档使用。

8.4.2　用户定义的模块属性

用户定义属性的语法如下：

- **-SomeTag**(Value).

SomeTag必须是一个原子，而Value必须是一个字面数据类型。模块属性的值会被编译进模块，可以在运行时提取。这个例子里的模块包含了一些用户定义的属性：

attrs.erl

```
-module(attrs).
-vsn(1234).
-author({joe,armstrong}).
-purpose("example of attributes").
-export([fac/1]).

fac(1) -> 1;
fac(N) -> N * fac(N-1).
```

可以用下面的方式提取这些值：

```
1> attrs:module_info().
[{exports,[{fac,1},{module_info,0},{module_info,1}]},
{imports,[]},
{attributes,[{vsn,[1234]},
             {author,[{joe,armstrong}]},
             {purpose,"example of attributes"}]},
             {compile,[{options,[]},
             {version,"4.8"},
             {time,{2013,5,3,7,36,55}},
             {source,"/Users/joe/jaerlang2/code/attrs.erl"}]}]
```

源代码文件所含的用户定义属性再一次出现了，它们表现为{attributes, ...}的下属数据类型。元组{compile, ...}包含了编译器添加的信息。{version,"4.8"}这个值是编译器的版本号，不应与模块属性里定义的vsn标签相混淆。在上面的例子里，attrs:module_info()返回一个属性列表，内含所有与被编译模块相关的元数据。attrs:module_info(X)（X可以是exports、imports、attributes和compile中的一个）会返回与模块相关的单个属性。

请注意，函数module_info/0和module_info/1会在模块编译时自动创建。

要运行attrs:module_info，必须先把attrs模块的beam代码加载到Erlang虚拟机里。也可以使用beam_lib模块来提取同样的信息，这样就不必载入attrs模块了。

```
3> beam_lib:chunks("attrs.beam",[attributes]).
{ok,{attrs,[{attributes,[{author,[{joe,armstrong}]},
                         {purpose,"example of attributes"},
                         {vsn,[1234]}]}]}}
```

beam_lib:chunks可以在不载入模块代码的情况下提取模块里的属性数据。

8.5 块表达式

块表达式用于以下情形：代码某处的Erlang语法要求单个表达式，但我们想使用一个表达式序列。举个例子，在一个形式为[E || ...]的列表推导中，语法要求E是单个表达式，但我们也许想要在E里做不止一件事情。

```
begin
   Expr1,
   ...,
   ExprN
end
```

你可以用块表达式归组一个表达式序列，就像子句的主体一样。begin ... end的值就是块里最后那个表达式的值。

8.6 布尔值

Erlang没有单独的布尔值类型。不过原子true和false具有特殊的含义，可以用来表示布尔值。

有时候编写的函数会返回两个可能的原子值中的一个。这时，正确的做法是确保它们返回一个布尔值。与此同时，让你的函数名称反映出它们会返回布尔值也是一个好主意。

举个例子，假设想编写一个程序来表示某个文件的状态，一来二去也许就落到一个返回open（打开）或closed（关闭）的file_state(File)函数身上。编写这个函数时，可以考虑重命名该函数并让它返回一个布尔值。稍加思索后，我们可以重新编写程序，让一个名为is_file_open(File)的函数返回true或false。

表示状态时用布尔值而非另选两个原子的原因很简单。标准库里有大量函数作用于返回布尔值的函数。因此，如果确保所有的函数都返回布尔值，就能将它们用于标准库函数了。

举个例子，假设有一个文件列表L并想把它分成一个打开文件列表和一个关闭文件列表。可以编写以下代码来利用标准库：

```
lists:partition(fun is_file_open/1, L)
```

但如果用的是file_state/1函数，恐怕就要先编写一个转换程序才能调用库方法了。

```
lists:partition(fun(X) ->
                  case file_state(X) of
                      open   -> true;
                      closed -> false
                    end, L)
```

8.7　布尔表达式

可用的布尔表达式有四种。

- not B1：逻辑非
- B1 and B2：逻辑与
- B1 or B2：逻辑或
- B1 xor B2：逻辑异或

在所有这些表达式里，B1和B2都必须是布尔值或者执行结果为布尔值的表达式。这里有一些例子：

```
1> not true.
false
2> true and false.
false
3> true or false.
true
4> (2 > 1) or (3 > 4).
true
```

8.8　字符集

从Erlang的R16B版开始，Erlang源代码文件都假定采用UTF-8字符集编码。在这之前用的是ISO-8859-1（Latin-1）字符集。这就意味着所有UTF-8可打印字符都能在源代码文件里使用，无需使用任何转义序列。

Erlang内部没有字符数据类型。字符串其实并不存在，而是由整数列表来表示。用整数列表表示Unicode字符串是毫无问题的。

8.9　注释

Erlang里的注释从一个百分号字符（%）开始，一直延伸到行尾。Erlang没有块注释。

注意　*在代码示例里经常出现两个百分号字符（%%）。双百分号标记能被Erlang模式的Emacs编辑器识别，并启动注释行自动缩进功能。*

```
% 这是一段注释
my_function(Arg1, Arg2) ->
    case f(Arg1) of
        {yes, X} ->   % 如果成功
          ..
```

8.10　动态代码载入

动态代码载入是内建于Erlang核心的最惊人特性之一。它的美妙之处在于你无需了解后台的运作就能顺利实现它。

它的思路很简单：每当调用someModule:someFunction(...)时，调用的总是最新版模块里的最新版函数，哪怕当代码在模块里运行时重新编译了该模块也是如此。

如果在a循环调用b时重新编译了b，那么下一次a调用b时就会自动调用新版的b。如果有许多不同进程正在运行而它们都调用了b，那么当b被重新编译后，所有这些进程就都会调用新版的b。为了了解它的工作原理，我们将编写两个小模块：a和b。b模块非常简单。

b.erl

```
-module(b).
-export([x/0]).

x() -> 1.
```

现在来编写a。

a.erl

```
-module(a).
-compile(export_all).

 start(Tag) ->
    spawn(fun() -> loop(Tag) end).

loop(Tag) ->
    sleep(),
    Val = b:x(),
    io:format("Vsn1 (~p) b:x() = ~p~n",[Tag, Val]),
    loop(Tag).

sleep() ->
    receive
        after 3000 -> true
    end.
```

现在可以编译a和b，然后启动两个a进程。

```
1> c(b).
{ok, b}
2> c(a).
{ok, a}
3> a:start(one).
<0.41.0>
```

```
Vsn1 (one) b:x() = 1
4> a:start(two).
<0.43.0>
Vsn1 (one) b:x() = 1
Vsn1 (two) b:x() = 1
Vsn1 (one) b:x() = 1
Vsn1 (two) b:x() = 1
```

这些a进程休眠3秒钟后唤醒并调用`b:x()`，然后打印出结果。现在进入编辑器，把模块b改成下面这样：

```
-module(b).
-export([x/0]).

x() -> 2.
```

然后在shell里重新编译b。这是现在所发生的：

```
5> c(b).
{ok,b}
Vsn1 (one) b:x() = 2
Vsn1 (two) b:x() = 2
Vsn1 (one) b:x() = 2
Vsn1 (two) b:x() = 2
...
```

两个原版的a仍然在运行，但现在它们调用了*新版*的b。所以，在模块a里调用`b:x()`时，实际上是在调用“b的最新版”。我们可以随心所欲地多次修改并重新编译b，而所有调用它的模块无需特别处理就会自动调用新版的b。

8

现在已经重新编译了b，那么如果我们修改并重新编译a会发生什么？来做个试验，把a改成下面这样：

```
-module(a).
-compile(export_all).

start(Tag) ->
    spawn(fun() -> loop(Tag) end).

loop(Tag) ->
    sleep(),
    Val = b:x(),
    io:format("Vsn2 (~p) b:x() = ~p~n",[Tag, Val]),
    loop(Tag).

sleep() ->
    receive
        after 3000 -> true
    end.
```

现在编译并启动a。

```
6> c(a).
{ok,a}
Vsn1 (one) b:x() = 2
Vsn1 (two) b:x() = 2
...
7> a:start(three).
<0.53.0>
Vsn1 (one) b:x() = 2
Vsn1 (two) b:x() = 2
Vsn2 (three) b:x() = 2
Vsn1 (one) b:x() = 2
Vsn1 (two) b:x() = 2
Vsn2 (three) b:x() = 2
...
```

有趣的事情发生了。启动新版的a后，我们看到了新版正在运行。但是，那些运行最初版a的现有进程仍然在正常地运行旧版的a。

现在可以试着再次修改b。

```
-module(b).
-export([x/0]).

x() -> 3.
```

我们将在shell里重新编译b，观察会发生什么。

```
8> c(b).
{ok,b}
Vsn1 (one) b:x() = 3
Vsn1 (two) b:x() = 3
Vsn2 (three) b:x() = 3
...
```

现在新旧版本的a都调用了b的最新版。

最后，再次修改a（这是第三次修改a了）。

```
-module(a).
-compile(export_all).

start(Tag) ->
    spawn(fun() -> loop(Tag) end).

loop(Tag) ->
    sleep(),
    Val = b:x(),
    io:format("Vsn3 (~p) b:x() = ~p~n",[Tag, Val]),
    loop(Tag).

sleep() ->
    receive
        after 3000 -> true
    end.
```

现在，当我们重新编译a并启动一个新版的a时，就会看到以下输出：

```
9> c(a).
{ok,a}
Vsn2 (three) b:x() = 3
...
10> a:start(four).
<0.106.0>
Vsn2 (three) b:x() = 3
Vsn3 (four) b:x() = 3
Vsn2 (three) b:x() = 3
Vsn3 (four) b:x() = 3
...
```

这段输出里的字符串是由两个最新版本的a（第2版和第3版）生成的，而那些运行第1版a代码的进程已经消失了。

在任一时刻，Erlang允许一个模块的两个版本同时运行：当前版和旧版。重新编译某个模块时，任何运行旧版代码的进程都会被终止，当前版成为旧版，新编译的版本则成为当前版。可以把这想象成一个带有两个版本代码的移位寄存器。当添加新代码时，最老的版本就被清除了。一些进程可以运行旧版代码，与此同时，另一些则可以运行新版代码。

更多细节请参阅purge_module的文档[①]。

8.11 Erlang 的预处理器

Erlang模块在编译前会自动由Erlang的预处理器进行处理。预处理器会展开源文件里所有的宏，并插入必要的包含文件。

通常情况下，无需查看预处理器的输出，但在特定情形下（比如调试某个有问题的宏时），应该保存预处理器的输出。要查看some_module.erl模块的预处理结果，可以在操作系统的shell里输入以下命令。

```
$ erlc -P some_module.erl
```

这会生成一个名为some_module.P的清单文件。

8.12 转义序列

可以在字符串和带引号的原子里使用转义序列来输入任何不可打印的字符。表4列出了所有可用的转义序列。

让我们在shell里举一些例子来展示这些约定方式是如何工作的。（注意：格式字符串里的~w是指忠实地打印列表，而不对输出结果进行美化。）

① 最新链接为：http://www.erlang.org/doc/man/erlang.html#purge_module-1。

```
%% 控制字符
1> io:format("~w~n", ["\b\d\e\f\n\r\s\t\v"]).
[8,127,27,12,10,13,32,9,11]
ok
%% 字符串里的八进制字符
2> io:format("~w~n", ["\123\12\1"]).
[83,10,1]
ok
%% 字符串里的引号和反斜杠
3> io:format("~w~n", ["\'\"\\"]).
[39,34,92]
ok
%% 字符编码
4> io:format("~w~n", ["\a\z\A\Z"]).
[97,122,65,90]
ok
```

表4　转义序列

转义序列	含　　义	整数编码
`\b`	退格符	8
`\d`	删除符	127
`\e`	换码符	27
`\f`	换页符	12
`\n`	换行符	10
`\r`	回车符	13
`\s`	空格符	32
`\t`	制表符	9
`\v`	垂直制表符	11
`\x{...}`	十六进制字符（...是十六进制字符）	
`\^a..\^z`或`\^A..\^Z`	Ctrl+A至Ctrl+Z	1至26
`\'`	单引号	39
`\"`	双引号	34
`\\`	反斜杠	92
`\C`	C的ASCII编码（C是一个字符）	（一个整数）

8.13　表达式和表达式序列

在Erlang里，任何可以执行并生成一个值的事物都被称为表达式（expression）。这就意味着`catch`、`if`和`try...catch`这些都是表达式。而记录声明和模块属性这些不能被求值，所以它们不是表达式。

表达式序列（expression sequence）是一系列由逗号分隔的表达式。它们在`->`箭头之后随处可见。表达式序列`E1, E2,..., En`的值被定义为序列最后那个表达式的值，而该表达式在计算时可以使用`E1, E2`等表达式所创建的绑定。它就等价于LISP里的`progn`。

8.14 函数引用

我们有时想引用在当前或外部模块里定义的某个函数，可以用下列标记实现。

- fun LocalFunc/Arity
  用于引用当前模块里参数为Arity的本地函数LocalFunc。

- fun Mod:RemoteFunc/Arity
  用于引用Mod模块里参数为Arity的外部函数RemoteFunc。
  这里有一个引用当前模块函数的例子：

```
-module(x1).
-export([square/1, ...]).

square(X) -> X * X.
...
double(L) -> lists:map(fun square/1, L).
```

如果要调用某个外部模块里的函数，就可以像下面这个例子一样引用它：

```
-module(x2).
...
double(L) -> lists:map(fun x1:square/1, L).
```

fun x1:square/1指的是x1模块里的square/1函数。

请注意，包含模块名的函数引用提供了动态代码升级的切换点。更多细节请参阅8.10节。

8

8.15 包含文件

下面的语法可以用来包含文件：

```
-include(Filename).
```

按照Erlang的惯例，包含文件的扩展名是.hrl。FileName应当包含一个绝对或相对路径，使预处理器能找到正确的文件。包含库的头文件（library header file）时可以用下面的语法：

```
-include_lib(Name).
```

这里有一个例子：

```
-include_lib("kernel/include/file.hrl").
```

在这种情况下，Erlang编译器会找到正确的包含文件。（前面例子中的kernel是指定义该头文件的应用。）

包含文件里经常会有记录的定义。如果许多模块需要共享通用的记录定义，就会把它们放到包含文件里，再由所有需要这些定义的模块包含此文件。

8.16　列表操作：++和--

++和--是用于列表添加和移除的中缀操作符。

A ++ B使A和B相加（也就是附加）。

A -- B从列表A中移除列表B。移除的意思是B中所有元素都会从A里面去除。请注意：如果符号X在B里只出现了K次，那么A只会移除前K个X。

这里有一些例子：

```
1> [1,2,3] ++ [4,5,6].
[1,2,3,4,5,6]
2> [a,b,c,1,d,e,1,x,y,1] -- [1].
[a,b,c,d,e,1,x,y,1]
3> [a,b,c,1,d,e,1,x,y,1] -- [1,1].
[a,b,c,d,e,x,y,1]
4> [a,b,c,1,d,e,1,x,y,1] -- [1,1,1].
[a,b,c,d,e,x,y]
5> [a,b,c,1,d,e,1,x,y,1] -- [1,1,1,1].
[a,b,c,d,e,x,y]
```

++也可以用在模式里。在匹配字符串时，可以编写如下模式：

```
f("begin" ++ T) -> ...
f("end" ++ T) -> ...
...
```

子句1里的模式会扩展成[$b,$e,$g,$i,$n|T]。

8.17　宏

Erlang的宏以如下方式编写：

```
-define(Constant, Replacement).
-define(Func(Var1, Var2,.., Var), Replacement).
```

当Erlang的预处理器epp碰到一个?MacroName形式的表达式时，就会展开这个宏。宏定义里出现的变量会匹配对应宏调用位置的完整形式。

```
-define(macro1(X, Y), {a, X, Y}).

foo(A) ->
    ?macro1(A+10, b)
```

它展开后是这样的：

```
foo(A) ->
    {a,A+10,b}.
```

另外还有一些预定义宏提供了关于当前模块的信息。列举如下：

- ?FILE展开成当前的文件名；

- ❑ ?MODULE展开成当前的模块名；
- ❑ ?LINE展开成当前的行号。

宏控制流

模块的内部支持下列指令，可以用它们来控制宏的展开。

- -undef(Macro).
  取消宏的定义，此后就无法调用这个宏了。

- -ifdef(Macro).
  仅当Macro有过定义时才执行后面的代码。

- -ifndef(Macro).
  仅当Macro未被定义时才执行后面的代码。

- -else.
  可用于ifdef或ifndef语句之后。如果条件为否，else后面的语句就会被执行。

- -endif.
  标记ifdef或ifndef语句的结尾。

条件宏必须有恰当的嵌套。按惯例它们被归组如下：

```
-ifdef(<FlagName>).
-define(...).
-else.
-define(...).
-endif.
```

我们可以用这些宏来定义一个DEBUG（调试）宏。这里是一个例子：

m1.erl

```
-module(m1).
-export([loop/1]).

-ifdef(debug_flag).
-define(DEBUG(X), io:format("DEBUG ~p:~p ~p~n",[?MODULE, ?LINE, X])).
-else.
-define(DEBUG(X), void).
-endif.

loop(0) ->
    done;
loop(N) ->
    ?DEBUG(N),
    loop(N-1).
```

> **注意** io:format(String, [Args])会根据String里的格式信息在Erlang shell中打印出[Args]所含的变量。格式编码用一个~符号作为前缀。~p是美化打印（pretty print）的简称，~n则会生成一个新行。io:format能理解的格式选项数量极其庞大，更多信息请参见16.3.1节。

为了启用这个宏，我们在编译代码时设置了debug_flag。具体做法是给c/2添加一个额外参数如下：

```
1> c(m1, {d, debug_flag}).
{ok,m1}
2> m1:loop(4).
DEBUG m1:13 4
DEBUG m1:13 3
DEBUG m1:13 2
DEBUG m1:13 1
done
```

如果没有设置debug_flag，这个宏就只会展开成原子void。选择这个名称没有什么实际意义，只是用来提醒你没有人会对这个宏的值感兴趣。

8.18　模式的匹配操作符

假设有如下代码：

```
Line 1 func1([{tag1, A, B}|T]) ->
     2     ...
     3     ... f(..., {tag1, A, B}, ...)
     4     ...
```

我们在第1行模式匹配了数据类型{tag1, A, B}，在第3行用参数{tag1, A, B}调用了f。这么做时，系统会重建数据类型{tag1, A, B}。一种更高效且不易出错的方式是把这个模式指派给一个临时变量Z，然后将它传递给f，就像这样：

```
func1([{tag1, A, B}=Z|T]) ->
    ...
    ... f(... Z, ...)
    ...
```

匹配操作符可以用在模式里的任何位置，因此如果有两个需要重建的数据类型，比如下面代码里的：

```
func1([{tag, {one, A}, B}|T]) ->
   ...
   ...  f(..., {tag, {one,A}, B}, ...),
   ...  g(..., {one, A}), ...)
   ...
```

就可以引入两个新变量Z1和Z2，像下面这样写：

```
func1([{tag, {one, A}=Z1, B}=Z2|T]) ->
   ..,.
   ...  f(..., Z2, ...),
   ...  g(..., Z1, ...),
   ...
```

8.19 数字

Erlang里的数字不是整数就是浮点数。

8.19.1 整数

整数的运算是精确的，而且用来表示整数的位数只受限于可用的内存。

整数可以有三种不同的写法。

- *传统写法*
  在这里，整数的写法和你预料的一样。比如，12、12375和-23427都是整数。

- *K进制整数*
  除10以外的数字进制整数使用K#Digits这种写法。因此，可以把一个二进制数写成2#00101010，或者把一个十六进制数写成16#af6bfa23。对大于10的进制而言，abc...（或ABC...）这些字符代表了数字10、11和12，以此类推。最高的进制数是36。

- *$写法*
  $C这种写法代表了ASCII字符C的整数代码。因此，$a是97的简写，$1是49的简写，以此类推。

还可以紧挨着$使用表4里描述的任意转义序列。所以，$\n是10，$\^c是3，以此类推。

这里有一些整数的例子：

```
0 -65 2#010001110 -8#377 16#fe34 16#FE34 36#wow
```

它们的值分别是0、-65、142、-255、65076、65076和42368。

8.19.2 浮点数

一个浮点数由五部分组成：一个可选的正负号，一个整数部分，一个小数点，一个分数部分和一个可选的指数部分。

这里有一些浮点数的例子：

```
1.0 3.14159 -2.3e+6 23.56E-27
```

解析后的浮点数在系统内部使用IEEE 754的64位格式表示。绝对值在10^{-323}到10^{308}范围内的实数可以用Erlang的浮点数表示。

8.20　操作符优先级

表5按照降序的优先级展示了所有Erlang操作符和它们的结合性。操作符优先级和结合性用于确定无括号表达式的执行顺序。

表5　操作符优先级

操　作　符	结　合　性
:	
#	
（一元）+、（一元）-、bnot、not	
/、*、div、rem、band、and	左结合
+、-、bor、bxor、bsl、bsr、or、xor	左结合
++、--	右结合
==、/=、=<、<、>=、>、=:=、=/=	
andalso	
orelse	
= !	右结合
catch	

优先级更高（在表格更上方）的表达式会首先执行，然后才轮到优先级较低的表达式。举个例子，要执行3+4*5+6，首先会执行子表达式4*5，因为（*）在表中的位置高于（+）。现在，待执行的表达式变成3+20+6。又因为（+）是一个左结合操作符，我们将它解读为(3+20)+6，所以会首先执行3+20，得23，最后再执行23+6。

3+4*5+6用完整的括号形式表现就是((3+(4*5))+6)。和所有编程语言一样，与其单纯依赖这些优先级规则，不如用括号来标明范围。

8.21　进程字典

每个Erlang进程都有一个被称为进程字典（process dictionary）的私有数据存储区域。进程字典是一个关联数组（在其他语言里可能被称作map、hashmap或者散列表），它由若干个键和值组成。每个键只有一个值。

这个字典可以用下列内置函数进行操作。

- put(Key, Value) -> OldValue.

  给进程字典添加一个Key, Value组合。put的值是OldValue，也就是Key之前关联的值。如果没有之前的值，就返回原子undefined。

- get(Key) -> Value.

  查找Key的值。如果字典里存在Key，Value组合就返回Value，否则返回原子undefined。

- get() -> [{Key,Value}].

  返回整个字典，形式是一个由{Key,Value}元组所构成的列表。

- get_keys(Value) -> [Key].

  返回一个列表，内含字典里所有值为Value的键。

- erase(Key) -> Value.

  返回Key的关联值，如果不存在则返回原子undefined。最后，删除Key的关联值。

- erase() -> [{Key,Value}].

  删除整个进程字典。返回值是一个由{Key,Value}元组所构成的列表，代表了字典删除之前的状态。

这里有一个例子：

```
1> erase().
[]
2> put(x, 20).
undefined
3> get(x).
20
4> get(y).
undefined
5> put(y, 40).
undefined
6> get(y).
40
7> get().
[{y,40},{x,20}]
8> erase(x).
20
9> get().
[{y,40}]
```

如你所见，进程字典里的变量和命令式编程语言里传统可变变量的行为非常相似。如果使用进程字典，你的代码就不再是无副作用的了，而且我们在3.3.1节讨论过的所有使用非破坏性变量的好处也将不复存在。出于这个原因，应当少用进程字典。

注意 我很少使用进程字典。进程字典可能会给你的程序引入不易察觉的bug，让调试变得困难。但是有一种用法我是支持的，那就是用进程字典来保存“一次性写入”的变量。如果某个键一次性获得一个值而且不会改变它，那么将其保存在进程字典里在某些时候还是可以接受的。

8.22　引用

引用（reference）是一种全局唯一的Erlang数据类型。它们由内置函数`erlang:make_ref()`创建。引用的用途是创建独一无二的标签，把它存放在数据里并在后面用于比较是否相等。举个例子，bug跟踪系统可以给每个新的bug报告添加一个引用，从而赋予它一个独一无二的标识。

8.23　短路布尔表达式

短路布尔表达式（short-circuit boolean expression）是一种只在必要时才对参数求值的表达式。“短路”布尔表达式有两种。

- Expr1 orelse Expr2

  它会首先执行`Expr1`。如果`Expr1`的执行结果是`true`，`Expr2`就不再执行。如果`Expr1`的执行结果是`false`，则会执行`Expr2`。

- Expr1 andalso Expr2

  它会首先执行`Expr1`。如果`Expr1`的执行结果是`true`，则会执行`Expr2`。如果`Expr1`的执行结果是`false`，`Expr2`就不再执行。

注意　在对应的布尔表达式里（`A or B`和`A and B`），两边的参数总会被执行，即使表达式的真值只需要第一个表达式的值就能确定也是如此。

8.24　比较数据类型

表6列出了全部8种可用的数据类型比较操作。

表6　比较数据类型

操作符	含　义
X > Y	X大于Y
X < Y	X小于Y
X =< Y	X等于或小于Y
X >= Y	X大于或等于Y
X == Y	X等于Y
X /= Y	X不等于Y
X =:= Y	X与Y完全相同
X =/= Y	X与Y不完全相同

为了便于比较，我们给所有的数据类型做了全排序（total ordering）的定义。定义的结果如下：

number < atom < reference < fun < port < pid < tuple (and record) < map < list < binary

比如，根据定义，一个数字（任何数字）小于一个原子（任何原子），而一个元组大于一个原子，以此类推。（请注意，出于排序的需要，端口和进程标识符也被包括在这个名单里。）

有了所有数据类型的全排序就意味着可以对任何类型的列表进行排序，以及根据键的排序顺序构建高效的数据访问方式。

所有的数据类型比较操作符（除了=:=和=/=）在参数全为数字时具有以下行为。

- 如果一个参数是整数而另一个是浮点数，那么整数会先转换成浮点数，然后再进行比较。
- 如果两个参数都是整数或者都是浮点数，就会"按原样"使用，也就是不做转换。

还应当非常谨慎地使用==（特别是对C或Java程序员而言），100次里有99次应该用的是=:=。==只有在比较浮点数和整数时才有用。=:=则是用来测试两个数据类型是否完全相同。

完全相同的意思是具有相同的值（类似Common Lisp的EQUAL）。因为值是不可变的，所以这里不涉及任何指针标识。如果不确定的话，就用=:=，见到==时则要三思。请注意，刚才的注解也适用于/=和=/=，/=的意思是"不等于"，而=/=的意思是"不完全相同"。

注意　你会在很多库代码和已发布代码里看到该用=:=的地方用了==操作符。幸运的是，这类错误通常不会导致程序出错，因为如果==的参数不包含任何浮点数的话，那么这两个操作符的行为就是相同的。

还应该知道，函数的子句匹配总是意味着精确的模式匹配，所以如果定义了一个fun `F = fun(12) -> ... end`，那么试图执行`F(12.0)`就会出错。

8

8.25　元组模块

调用`M:f(Arg1, Arg2, ..., ArgN)`时，我们假定M是一个模块名。但M也可以是一个形式为`{Mod1, X1, X2, ... Xn}`的元组。在这种情况下，调用的函数就是`Mod1:f(Arg1, Arg2, ..., Arg3, M)`。

这种机制可以用来创建"有状态的模块"（将在24.3节里讨论）和"适配器模式"（将在24.4节里讨论）。

8.26　下划线变量

关于变量还有一件事需要说一下。`_VarName`这种特殊语法代表一个常规变量（normal variable），而不是匿名变量。一般来说，当某个变量在子句里只使用了一次时，编译器会生成一个警告，因为这通常是出错的信号。但如果这个只用了一次的变量以下划线开头，就不会有错误消息。

因为`_Var`是常规变量，所以如果你忘了这一点并将它用于"不关心"的模式，就可能导致非常微妙的bug。举个例子，在一个非常复杂的模式匹配里，也许很难察觉`_Int`被多次不恰当地

使用，从而导致模式匹配失败。

下划线变量有两种主要的用途。

- 命名一个我们不打算使用的变量。例如，相比open(File, _)，open(File, _Mode)这种写法能让程序的可读性更高。
- 用于调试。举个例子，假设编写如下代码：

```
some_func(X) ->
    {P, Q} = some_other_func(X),
    io:format("Q = ~p~n", [Q]),
    P.
```

编译它不会产生错误消息。

现在注释掉下面的格式语句：

```
some_func(X) ->
    {P, Q} = some_other_func(X),
    %% io:format("Q = ~p~n", [Q]),
    P.
```

如果编译它，编译器就会生成一个变量Q未使用的警告。

如果把这个函数重新编写如下：

```
some_func(X) ->
    {P, _Q} = some_other_func(X),
    io:format("_Q = ~p~n", [_Q]),
    P.
```

就可以注释掉格式语句，而编译器也不会再抱怨了。

现在我们真正完成了Erlang的顺序编程部分。

在接下来的两章里我们将完成本书的第二部分。我们会从描述Erlang函数类型的类型表示法开始，然后介绍一些可以用来对Erlang代码做类型检查的工具。该部分的最后一章将介绍几种编译和运行程序的方法。

8.27　练习

(1) 复习这一章里关于Mod:module_info()的部分。输入命令dict:module_info()。这个模块返回了多少个函数？

(2) code:all_loaded()命令会返回一个由{Mod,File}对构成的列表，内含所有Erlang系统载入的模块。使用内置函数Mod:module_info()了解这些模块。编写一些函数来找出哪个模块导出的函数最多，以及哪个函数名最常见。编写一个函数来找出所有不带歧义的函数名，也就是那些只在一个模块里出现过的函数名。

第9章 类型

Erlang有一种类型表示法，它可以用来定义新的数据类型并给代码添加类型注解（type annotation）。类型注解能让代码更易理解和维护，还可以在编译时检测错误。

在这一章里，我们将介绍类型表示法，并讨论两个可以用来寻找代码错误的程序。

将要讨论的这两个程序名为`dialyzer`和`typer`，它们存在于标准Erlang分发套装里。dialyzer代表“DIscrepancy AnaLYZer for ERlang programs”（用于Erlang程序的差错分析器），它的作用名副其实：寻找Erlang代码里的差错。`typer`提供了程序里使用的类型信息。dialyzer和typer不需要类型注解就能完美工作，但如果你给程序添加了类型注解，这些工具的分析质量就会变得更高。

这一章比较复杂，所以我们将从一个简单的例子着手，然后深入一些，看看类型的语法。在此之后，会进入dialyzer的教程，讨论使用dialyzer时应采用的工作流，以及dialyzer无法找到的错误类型。最后，通过介绍dialyzer的工作原理来更好地理解它所找到的错误。

9.1 指定数据和函数类型

我们即将开始一段漫长的徒步旅程，幸运的是有一个模块可以让我们来规划一番。此模块的开头部分是这样的：

walks.erl

```
-module(walks).
-export([plan_route/2]).

-spec plan_route(point(), point()) -> route().
-type direction() :: north | south | east | west.
-type point()     :: {integer(), integer()}.
-type route()     :: [{go,direction(),integer()}].

…
```

这个模块导出了一个名为`plan_route/2`的函数。该函数的输入和返回类型由一个类型规范（type specification）指定，类型声明（type declaration）里还定义了三个新的类型。对它们的解读如下：

- -spec plan_route(point(), point()) -> route().
  它的意思是如果调用plan_route/2函数时使用了两个类型为point()的参数，此函数就会返回一个类型为route()的对象。

- -type direction() :: north | south | east | west.
  引入一个名为direction()的新类型，它的值是下列原子之一：north、south、east或west。

- -type point() :: {integer(), integer()}.
  指point()类型是一个包含两个整数的元组（integer()是预定义类型）。

- -type route() :: [{go, direction(), integer()}].
  将route()类型定义为一个由三元组（3-tuple）构成的列表，每个元组都包含一个原子go，一个类型为direction的对象和一个整数。[X]这种表示法的意思是一个由X类型构成的列表。

单凭类型注解我们就能想象到执行plan_route后会看到如下输出：

```
> walks:plan_route({1,10}, {25, 57}).
[{go, east, 24},
 {go, north, 47},
 ...
 ]
```

当然，我们完全不知道plan_route函数到底会不会返回，也许它直接崩溃了，不会返回任何值。但如果它真的返回了一个值，那么当输入参数的类型是point()时，返回值的类型应该是route()。同样不知道之前那个表达式里的数字有什么含义。它们是英里、公里、厘米还是其他？所知道的就是类型声明能告诉我们的，即它们都是整数。

为了增强类型的表达能力，可以用描述性变量给它们加上注解。举个例子，修改plan_route的规范如下：

```
-spec plan_route(From:: point(), To:: point()) -> ...
```

类型注解里的名称From（起点）和To（终点）让用户对这些参数在函数里扮演的角色有了一定的了解。它们还用来在文档里的名称和类型注解里的变量之间建立联系。Erlang的官方文档对于编写类型注解有着严格的规定，使类型注解里的名称能够对应文档里的名称。

声明我们的路径（route）从From开始并且From是一对整数可能足以描述该函数，也可能需要更多信息，这需要根据上下文确定。可以通过添加更多的信息来轻松改进类型定义。比如这么写：

```
-type angle()       :: {Degrees::0..360, Minutes::0..60, Seconds::0..60}.
-type position()    :: {latitude | longitude, angle()}.
-spec plan_route1(From::position(), To::position()) -> ...
```

这种新形式提供了很多的信息，但是仍然免不了让人猜测。我们可能会猜想角度的单位是度（degree），因为允许值的范围是0到360，但它们也可能是弧度，这样的话就猜错了。

随着类型注解变得越来越长，追求更加精确最终可能会以文字冗长作为代价。越来越庞大的注解可能会让代码变得难以阅读。编写好的代码注解和编写好的代码一样，也是一种艺术——它非常困难，需要多年的练习。它是参禅的一种形式：你做得越多，就越容易，做得也越好！

我们已经通过一个简单的例子看到了类型是如何定义的，下一节将正式介绍类型表示法。在此之后，我们将进入dialyzer的教程。

9.2 Erlang 的类型表示法

到目前为止，我们已经通过非正式的方式介绍了类型。要充分利用类型系统，需要理解类型的语法，这样才能阅读和编写更准确的类型描述。

9.2.1 类型的语法

类型定义可以使用以下的非正式语法：

```
T1 :: A | B | C ...
```

它的意思是T1被定义为A、B或C其中之一。

用这种表示法，可以定义一些Erlang类型如下：

```
Type :: any() | none() | pid() | port() | reference() | []
      | Atom | binary() | float() | Fun | Integer | [Type] |
      | Tuple | Union | UserDefined

Union :: Type1 | Type2 | ...

Atom :: atom() | Erlang_Atom

Integer :: integer() | Min .. Max

Fun :: fun() | fun((...) -> Type)

Tuple :: tuple() | {T1, T2, ... Tn}
```

在上面的例子中，any()是指任意Erlang数据类型，X()是指一个类型为X的Erlang对象，而none()标识则用来指代永不返回的函数类型。

[X]这种表示法指代一个由X类型构成的列表，{T1, T2, ..., Tn}指代一个大小为n，参数类型分别为T1, T2, ... Tn的元组。

定义新的类型可以使用以下语法：

```
-type NewTypeName(TVar1, TVar2, ... TVarN) :: Type.
```

TVar1至TVarN是可选的类型变量，Type是一个类型表达式。

这里有一些例子：

```
-type onOff()        :: on | off.
-type person()       :: {person, name(), age()}.
-type people()       :: [person()].
-type name()         :: {firstname, string()}.
-type age()          :: integer().
-type dict(Key,Val)  :: [{Key,Val}].
...
```

根据这些规则，{firstname, "dave"}属于name()类型，[{person, {firstname, "john"}, 35}, {person, {firstname,"mary"}, 26}]属于people()类型，以此类推。dict(Key,Val)类型展示了类型变量的用法，它把一个字典类型定义为由{Key, Val}元组构成的列表。

9.2.2 预定义类型

除了类型语法以外，还有下面这些预定义的类型别名：

```
-type term() :: any().
-type boolean() :: true | false.
-type byte() :: 0..255.
-type char() :: 0..16#10ffff.
-type number() :: integer() | float().
-type list() :: [any()].
-type maybe_improper_list() :: maybe_improper_list(any(), any()).
-type maybe_improper_list(T) :: maybe_improper_list(T, any()).
-type string() :: [char()].
-type nonempty_string() :: [char(),...].
-type iolist() :: maybe_improper_list(byte() | binary() | iolist(),
                                      binary() | []).
-type module() :: atom().
-type mfa() :: {atom(), atom(), atom()}.
-type node() :: atom().
-type timeout() :: infinity | non_neg_integer().
-type no_return() :: none().
```

maybe_improper_list用于指定带有非空（non-nil）最终列表尾的列表类型。这样的列表很少会用到，但是指定它们的类型是可能的！

还有少量其他的预定义类型。non_neg_integer()是一个非负的整数，pos_integer()是一个正整数，neg_integer()是一个负整数。最后，[X,...]这种表示法的意思是一个由X类型构成的非空列表。

现在我们能定义类型了，接下来介绍函数规范。

9.2.3 指定函数的输入输出类型

函数规范说明了某个函数的参数属于何种类型，以及该函数的返回值属于何种类型。函数规

范的编写方式如下：

```
-spec functionName(T1, T2, ..., Tn) -> Tret when
    Ti :: Typei,
    Tj :: Typej,
    ...
```

这里的T1, T2,..., Tn描述了某个函数的参数类型，Tret描述了函数返回值的类型。如果有必要，可以在可选关键字when后面引入额外的类型变量。

我们将从一个例子入手。下面这个类型规范：

```
-spec file:open(FileName, Modes) -> {ok, Handle} | {error, Why} when
    FileName  :: string(),
    Modes     :: [Mode],
    Mode      :: read | write | ...
    Handle    :: file_handle(),
    Why       :: error_term().
```

说明如果打开FileName文件，得到的返回值不是{ok, Handle}就是{error, Why}。FileName是一个字符串，Modes是一个由Mode组成的列表，而Mode是read、write等模式中的一个。

上面这个函数规范可以用多种等价的方式编写。举个例子，可以像下面这样不使用限定词when：

```
-spec file:open(string(), [read|write|...] -> {ok, Handle} | {error, Why}
```

这样做的问题在于：首先，失去了FileName和Modes这些描述性的变量；其次，类型规范的长度大大增加，导致阅读和在打印文档里格式化的难度增加。在理想情况下，程序的后面应该附有文档，而如果没有给函数的参数命名，就无法在文档里引用它们。

在第一种编写规范的方式下是这么写的：

```
-spec file:open(FileName, Modes) -> {ok, Handle} | {error, Why} when
    FileName  :: string(),
    ...
```

这个函数的任何文档都可以毫无歧义地引用打开的文件，方法是使用其名称FileName。如果丢弃了限定词when：

```
-spec file:open(string(), [read|write|...) -> {ok, Handle} | {error, Why}.
```

那么文档在引用打开的文档时就不得不称其为“open函数的第一个参数”，这种迂回的说法对第一种规范编写方式而言是不必要的。

类型变量可以在参数里使用，如下所示：

```
-spec lists:map(fun((A) -> B), [A]) -> [B].
-spec lists:filter(fun((X) -> bool()), [X]) -> [X].
```

它的意思是map函数接受一个从A类型到B类型的函数和一个由A类型对象组成的列表，然后

返回一个由B类型对象组成的列表，以此类推。

9.2.4　导出类型和本地类型

有时候我们希望某个类型的定义局限在该定义所属的模块内部，而在另一些情况下则希望把此类型导出至别的模块。想象一下有两个模块a和b。a模块生成rich_text类型的对象，b模块则操作这些对象。在a模块里加入以下注解：

```
-module(a).
-type rich_text() :: [{font(), char()}].
-type font()      :: integer().
-export_type([rich_text/0, font/0]).
...
```

我们不仅声明了一个富文本和一个字体类型，还用注解-export_type(...)导出了它们。

假设b模块能操作富文本的实例，比如内含一个计算富文本对象长度的rich_text_length函数。编写此函数的类型规范如下：

```
-module(b).
...
-spec rich_text_length(a:rich_text()) -> integer().
...
```

rich_text_length的输入参数使用了完全限定的类型名a:rich_text()，它是指从a模块导出的rich_text()类型。

9.2.5　不透明类型

在上一节里，a和b这两个模块通过操作富文本对象的内部结构相互协作。但是，我们也许希望隐藏富文本数据结构的内部细节，使得只有创建此数据结构的模块才了解类型的细节。可以通过一个例子来更好地理解这一点。

假设a模块的开头部分如下：

```
-module(a).
-opaque rich_text() :: [{font(), char()}].
-export_type([rich_text/0]).

-export([make_text/1, bounding_box/1]).
-spec make_text(string()) -> rich_text().
-spec bounding_box(rich_text()) -> {Height::integer(), Width::integer()}.
...
```

下面这个语句：

```
-opaque rich_text() :: [{font(), char()}].
```

创建了一个名为rich_text()的不透明类型（opaque type）。现在来看一些尝试操作富文本

对象的代码：

```
-module(b).
...

do_this() ->
   X = a:make_text("hello world"),
   {W, H} = a:bounding_box(X)
```

b模块永远不需要知道变量X的内部结构。X是在a模块里创建的，调用bounding_box(X)时会把它传回a。

现在假设我们编写了一段利用了某些rich_text对象外形知识的代码。比如，假设创建了一个富文本对象，然后询问需要什么字体来渲染此对象。我们也许会这么写：

```
-module(c).
...

fonts_in(Str) ->
    X = a:make_text(Str),
    [F || {F,_} <- X].
```

在列表推导里，我们“知道”X是一个由双元组构成的列表，而在a模块里声明过make_text的返回类型是不透明的，意思是我们不该知道此类型的任何内部结构信息。利用此类型的内部结构信息被称为*抽象违规*（abstraction violation），如果正确声明了相关函数的类型可见性，这一违规就可以被dialyzer检测出来。

9.3　dialyzer 教程

第一次运行dialyzer时，需要为打算使用的所有标准库类型建立缓存。这个操作只需进行一次。启动dialyzer后它会告诉你怎么做。

```
$ dialyzer
Checking whether the PLT /Users/joe/.dialyzer_plt is up-to-date...
dialyzer: Could not find the PLT: /Users/joe/.dialyzer_plt
Use the options:
--build_plt   to build a new PLT; or
--add_to_plt  to add to an existing PLT

For example, use a command like the following:
dialyzer --build_plt --apps erts kernel stdlib mnesia
...
```

PLT是Persistent Lookup Table（持久性查询表）的缩写。PLT应当包含标准系统里所有类型的缓存。生成PLT需要花费几分钟的时间。我们输入的第一个命令会生成erts、stdlib和kernel的PLT。

```
$ dialyzer --build_plt --apps erts kernel stdlib
Compiling some key modules to native code... done in 0m59.78s
Creating PLT /Users/joe/.dialyzer_plt ...
Unknown functions:
compile:file/2
compile:forms/2
compile:noenv_forms/2
compile:output_generated/1
crypto:des3_cbc_decrypt/5
crypto:start/0
Unknown types:
compile:option/0
done in 4m3.86s
done (passed successfully)
```

现在PLT已经建成，我们准备好运行dialyzer了。出现未知函数的警告是因为列出的函数存在于外部的应用中，不属于我们选择进行分析的这三个。

dialyzer很很保守。如果它抱怨了，那就说明程序里确实存在着不一致性。制作dialyzer项目的一个目标就是消除虚假警告消息，即并非针对真正错误的警告消息。

在接下来的几节里，我们会给出一些错误程序的例子，对这些程序运行dialyzer，并展示dialyzer能够报告哪些类型的错误。

9.3.1　错误使用内置函数的返回值

dialyzer/test1.erl

```
-module(test1).
-export([f1/0]).

f1() ->
    X = erlang:time(),
    seconds(X).

seconds({_Year, _Month, _Day, Hour, Min, Sec}) ->
    (Hour * 60 + Min)*60 + Sec.
```

```
> dialyzer test1.erl
Checking whether the PLT /Users/joe/.dialyzer_plt is up-to-date... yes
Proceeding with analysis...
test1.erl:4: Function f1/0 has no local return
test1.erl:6: The call test1:seconds(X::
    {non_neg_integer(),non_neg_integer(),non_neg_integer()})
    will never return since it differs in the 1st argument
    from the success typing arguments: ({_,_,_,number(),number(),number()})
test1.erl:8: Function seconds/1 has no local return
test1.erl:8: The pattern {_Year, _Month, _Day, Hour, Min, Sec} can never
    match the type {non_neg_integer(),non_neg_integer(),non_neg_integer()}
done in 0m0.41s
```

这段相当吓人的错误消息出现的原因是erlang:time()返回一个名为{Hour, Min, Sec}的三元组，而不是我们期望的六元组。“Function f1/0 has no local return”（函数f1/0没有本地返回）的意思是f1/0会崩溃。dialyzer知道erlang:time()的返回值是{non_neg_integer(), non_neg_integer(), non_neg_integer()}类型的一个实例，因此绝不可能匹配seconds/1参数的六元组模式。

9.3.2　内置函数的错误参数

可以通过dialyzer来了解是否用错误的参数调用了内置函数。这方面的例子如下：

dialyzer/test2.erl

```
-module(test2).
-export([f1/0]).

f1() ->
    tuple_size(list_to_tuple({a,b,c})).
```

```
$ dialyzer test2.erl
test2.erl:4: Function f1/0 has no local return
test2.erl:5: The call erlang:list_to_tuple({'a','b','c'})
will never return since it differs in the 1st argument from the
success typing arguments: ([any()])
```

它告诉我们list_to_tuple期望的参数是[any()]类型的，而不是{'a','b','c'}。

9.3.3　错误的程序逻辑

dialyzer还可以检测出有问题的程序逻辑。这里有一个例子：

dialyzer/test3.erl

```
-module(test3).
-export([test/0, factorial/1]).

test() -> factorial(-5).

factorial(0) -> 1;
factorial(N) -> N*factorial(N-1).
```

```
$ dialyzer test3.erl
test3.erl:4: Function test/0 has no local return
test3.erl:4: The call test3:factorial(-5) will never return since
it differs in the 1st argument from the success typing
arguments: (non_neg_integer())
```

这其实是相当厉害的。阶乘（factorial）的定义有问题。如果用负数作为参数调用阶乘，这个程序就会进入无限循环，蚕食栈空间，最终Erlang会因为内存耗尽而崩溃。dialyzer根据阶乘的

参数属于`non_neg_integer()`类型推断出`factorial(-5)`这个调用是错误的。

dialyzer没有打印出它推测的函数类型，所以我们来问问`typer`。

```
$ typer test3.erl
-spec test() -> none().
-spec factorial(non_neg_integer()) -> pos_integer().
```

typer推测`factorial`的类型是`(non_neg_integer()) -> pos_integer()`，而`test()`的类型是`none()`。

这些程序的推理过程如下：递归的基本情形是`factorial(0)`，因此要让`factorial`的参数变成0，`factorial(N-1)`这个调用就必须最终降至0。因此N必须大于等于1，这就是阶乘类型的来由。这一点非常巧妙。

9.3.4　使用dialyzer

用dialyzer来检查程序里的类型错误需要按照一种特定的工作流进行。不要不加任何类型注解就编写完整个程序，然后当你感觉一切就绪后再回过头来到处添加类型注解并运行dialyzer。这么做，很可能会见到大量让人困惑的错误，从而不知道该从哪里着手修复错误。

使用dialyzer的最佳方式是将它用于每一个开发阶段。开始编写一个新模块时，*应该首先考虑类型*并声明它们，然后再编写代码。要为模块里所有的导出函数编写类型规范。先完成这一步再开始写代码。可以先注释掉未完成函数的类型规范，然后在实现函数的过程中取消注释。

现在可以开始编写函数了，一次写一个，每写完一个新函数就要用dialyzer检查一下，看看能否发现程序错误。如果函数是导出的，就加上类型规范；如果不是，就要考虑添加类型规范是否有助于类型分析或者能帮助我们理解程序（请记住，类型注解是很好的程序文档）。如果dialyzer发现了任何错误，就应该停下来思考并找出错误的准确含义。

9.3.5　干扰dialyzer的事物

dialyzer很容易受到干扰。我们可以通过遵循几条简单的规则来防止这种情况发生。

- 避免使用`-compile(export_all)`。如果导出了模块里的所有函数，dialyzer就可能无法推理出某些导出函数的参数，因为这些函数的调用位置和类型可能会千变万化。这些参数的值可能会扩散到模块的其他函数，导致让人困惑的错误。
- 为模块导出函数的所有参数提供详细的类型规范。尽量给导出函数的参数设置最严格的限制。举个例子，乍看之下你可能会推断函数的某个参数是一个整数，但经过进一步思考，也许就能确定该参数是一个正整数，甚至位于某个范围之内。把类型设置得越精确，dialyzer的分析结果就越出色。另外，如果可能，你还应该为代码添加精确的关卡测试。这会有助于程序分析，而且往往能帮助编译器生成质量更高的代码。
- 为记录定义里的所有元素提供默认的参数。如果不提供，原子`undefined`就会被当成默认值，而这个类型会逐渐扩散到程序的其他部分，可能导致奇特的类型错误。

- 把匿名变量用作函数的参数经常会导致结果类型不如你预想得那么精确。要尽可能地给变量添加限制。

9.4 类型推断与成功分型

dialyzer生成的某些错误十分奇特。要理解这些错误，必须理解dialyzer得出Erlang函数类型的过程。理解它能帮助我们解读这些古怪的错误消息。

类型推断（type inference）是指通过分析代码得出函数类型的过程。要做到这一点，我们会分析程序，寻找约束条件。用这些约束条件构建出一组约束方程式，然后求解。得到的一组类型就被称为此程序的成功分型（success typing）。来看一个简单的模块，看看它能告诉我们什么。

dialyzer/types1.erl

```
-module(types1).
-export([f1/1, f2/1, f3/1]).

f1({H,M,S}) ->
    (H+M*60)*60+S.
f2({H,M,S}) when is_integer(H) ->
    (H+M*60)*60+S.

f3({H,M,S}) ->
    print(H,M,S),
    (H+M*60)*60+S.

print(H,M,S) ->
    Str = integer_to_list(H) ++ ":" ++ integer_to_list(M) ++ ":" ++
        integer_to_list(S),
    io:format("~s", [Str]).
```

在阅读下一章之前，请花一点时间仔细观察上面的代码，试着找出代码里各个变量的类型。下面是运行dialyzer之后所发生的事：

```
$ dialyzer types1.erl
Checking whether the PLT /Users/joe/.dialyzer_plt is up-to-date... yes
Proceeding with analysis... done in 0m0.41s
done (passed successfully)
```

dialyzer在这段代码里没有找到类型错误。但这并不意味着代码就是正确的，它仅仅是指程序里所有数据类型的使用方式相互一致。我在把时、分、秒转换成秒时写的是`(H+M*60)*60+S`，而这是完全错误的，应该是`(H*60+M)*60+S`。没有任何类型系统能检测出这一点。所以即使程序具备正确的类型，仍然需要对其进行案例测试。

在这个程序上运行`typer`会产生以下输出：

```
$ typer types1.erl
%% File: "types1.erl"
%% ------------------
```

```
-spec f1({number(),number(),number()}) -> number().
-spec f2({integer(),number(),number()}) -> number().
-spec f3({integer(),integer(),integer()}) -> integer().
-spec print(integer(),integer(),integer()) -> 'ok'.
```

typer会报告它分析的模块里所有函数的类型。typer将函数f1的类型说明如下：

```
-spec f1({number(),number(),number()}) -> number().
```

这可以通过观察f1的定义得出，它的定义如下：

```
f1({H,M,S}) ->
   (H+M*60)*60+S.
```

这个函数为我们提供了5个不同的约束条件。首先，f1的参数必须是一个包含三个元素的元组。其次，每个算术操作符都增加了一个约束条件。比如，子表达式M*60告诉我们M必须属于number()类型，因为乘法操作符的左右参数都必须是数字。类似地，...+S也告诉我们S必须是一个数字。

现在来看看函数f2。下面列出了它的代码和推断类型：

```
f2({H,M,S}) when is_integer(H) ->
   (H+M*60)*60+S.

-spec f2({integer(),number(),number()}) -> number().
```

添加的关卡is_integer(H)是一个额外的约束条件，即H必须是一个整数。这个约束条件改变了f2元组参数里第一个元素的类型，把它从number()改成了更精确的integer()类型。

请注意，这里严谨的说法应该是“添加了额外的约束条件，因此，如果该函数能正常工作，H就必然是一个整数”。这就是我们把函数的推断类型称为合格类型的原因，从字面上讲就是“要让函数能成功执行，它的参数就必须属于这个类型”。

现在来关注types1.erl的最后一个函数。

```
f3({H,M,S}) ->
    print(H,M,S),
    (H+M*60)*60+S.

print(H,M,S) ->
    Str = integer_to_list(H) ++ ":" ++ integer_to_list(M) ++ ":" ++
          integer_to_list(S),
    io:format("~s", [Str]).
```

它的推断类型如下：

```
-spec f3({integer(),integer(),integer()}) -> integer().
-spec print(integer(),integer(),integer()) -> 'ok'.
```

这里你就能看到对integer_to_list的调用是如何把它的参数限制成一个整数的。随后，这个出现在print函数里的约束条件扩散到了f3函数的主体中。

如你所见，类型分析的过程有两个阶段。首先会得出一组约束方程式，然后进行求解。如果dialyzer没有找到错误，就表明这组约束方程式是有解的，typer则会打印出这些方程的解。如果这些方程式存在不一致性，无法求解，dialyzer就会报告一个错误。

现在，对之前的程序做一点小改动来引入一个错误，看看会对分析造成怎样的影响。

dialyzer/types1_bug.erl

```
-module(types1_bug).
-export([f4/1]).

f4({H,M,S}) when is_float(H) ->
    print(H,M,S),
    (H+M*60)*60+S.

print(H,M,S) ->
    Str = integer_to_list(H) ++ ":" ++ integer_to_list(M) ++ ":" ++
        integer_to_list(S),
    io:format("~s", [Str]).
```

首先运行typer。

```
$ typer types1_bug.erl
-spec f4(_) -> none().
-spec print(integer(),integer(),integer()) -> 'ok'.
```

typer报告说f4的返回类型是none()。这个特殊类型的意思是“此函数永远不会返回”。

运行dialyzer时会看到以下输出：

```
$ dialyzer types1_bug.erl
types1_bug.erl:4: Function f4/1 has no local return
types1_bug.erl:5: The call types1_bug:print(H::float(),M::any(),S::any())
   will never return since it differs in the 1st argument from the
   success typing arguments: (integer(),integer(),integer())
types1_bug.erl:8: Function print/3 has no local return
types1_bug.erl:9: The call erlang:integer_to_list(H::float())
   will never return since it differs in the 1st argument from the
   success typing arguments: (integer())
```

现在回过头来观察一下代码。关卡测试is_float(H)告诉系统H必然是一个浮点数。而随着H扩散到print函数，print内部的函数调用integer_to_list(H)却告诉系统H必然是一个整数。现在dialyzer无法确定这两种陈述哪一种才是正确的，因此假定它们都是错误的。这就是它报告“Function print/3 has no local return value”（print/3函数没有本地返回值）的原因。这是类型系统的局限性之一，它们能报告的就是程序存在不一致性，然后把问题留给程序员来分析解决。

9.5 类型系统的局限性

让我们来看一下给代码添加类型规范后会发生什么。我们将从众所周知的布尔函数and开

始。and只有在它的两个参数都为true时才为true，如果任何一个参数是false，它的值就是false。定义一个myand1函数（它的工作方式应该和and一样）如下：

types1.erl

```
myand1(true, true) -> true;
myand1(false, _)   -> false;
myand1(_, false)   -> false.
```

对它运行typer后可得到以下输出：

```
$typer types1.erl
-spec myand1(_,_) -> boolean().
...
```

myand1的推断类型是(_,_) -> boolean()，意思是myand1的各个参数都可以是任何你喜欢的值，返回的类型则是boolean。"myand1的参数可以是任何值"这个推断依据的是参数位置里的下划线。比如，myand1的子句2是myand1(false, _) -> false，基于此它推断出第二个参数可以是任何值。

现在，假设给这个模块添加一个错误的bug1函数如下：

types1.erl

```
bug1(X, Y) ->
    case myand1(X, Y) of
        true ->
            X + Y
    end.
```

然后让typer分析这个模块。

```
$ typer types1.erl
-spec myand1(_,_) -> boolean().
-spec bug1(number(), number()) -> number().
```

typer知道+用两个数字作为参数并返回一个数字，因此推断X和Y都是数字。它还推断出myand1的参数可以是任何值，这与X和Y都是数字不矛盾。如果在这个模块上运行dialyzer，它是不会返回错误的。typer认为用两个数字参数调用bug1会返回一个数字，但它不会。它会崩溃。这个例子展示了参数类型规范的不到位（即把_当作类型而非boolean()）会导致分析程序时无法发现的错误。

现在我们对类型的了解已经足够了。作为本书第二部分的结尾，我们将在下一章里介绍编译和运行程序的多种方式。我们能进行的shell操作里有很大一部分可以自动化进行，下一章将介绍几种实现方式。读完下一章后，你就具备了所有构建和运行顺序Erlang代码的知识。在此之后，就可以转向并发编程了。其实，并发编程才是本书的主题，但要跑必须先学会走路，要进行并发编程必须先学会顺序编程。

9.6　练习

(1) 编写一些导出单个函数的小模块，以及被导出函数的类型规范。在函数里制造一些类型错误，然后对这些程序运行dialyzer并试着理解错误消息。有时候你制造的错误无法被dialyzer发现，请仔细观察程序，找出没有得到预期错误的原因。

(2) 观察标准库代码里的类型注解。找到`lists.erl`模块的源代码并阅读里面所有的类型注解。

(3) 为什么在编写模块前需要先思考里面函数的类型？它是否在任何情况下都是一个好主意？

(4) 对不透明类型做些试验。创建两个模块，第一个模块导出一个不透明类型，第二个模块以能导致抽象违规的方式使用该不透明类型的内部数据结构。在这两个模块上运行dialyzer并确保理解了错误消息。

第 10 章 编译和运行程序

我们在前面几章里并没有过多谈及如何编译和运行程序，所用的都是Erlang shell。这对小例子来说足够了，但随着程序变得越来越复杂，你会希望这个过程能自动化来让自己轻松一点。这就要用到makefile。

事实上，有三种不同的方式可以运行程序。在这一章里，我们将完整介绍它们，这样你就可以根据具体情况选择最佳的方式了。

有时候一些问题会接连出现：makefile会失败，环境变量会出错，你的搜索路径也可能不正确。我们会帮助你应对这些状况，介绍出错时应该做些什么。

10.1 改变开发环境

开始用Erlang编程时，多半会把所有的模块和文件放在同一个目录里，然后从这个目录启动Erlang。如果这么做，Erlang的载入器可以百分百找到你的代码。但是，随着程序变得越来越复杂，你会想要把它们分成多个易于管理的区块，并把代码放进不同的目录里。另外，当你想要包含来自其他项目的代码时，这些外部代码会有自己的目录结构。

10.1.1 设置载入代码的搜索路径

Erlang的运行时系统使用一种代码自动载入机制。要让它能正确工作，必须设置一些搜索路径来找到正确版本的代码。

这种代码自动载入机制实际上是用Erlang编写的，之前在8.10节里曾经介绍过。代码的载入是“按需进行”的。

当系统尝试调用的函数属于一个尚未加载的模块时，就会出现一个异常，系统会尝试寻找缺失模块的对象代码文件。如果缺失的模块名为`myMissingModule`，代码载入器就会在当前载入路径的所有目录里搜索一个名为`myMissingModule.beam`的文件。只要找到相符的文件，搜索就会停止，此文件的目标代码会被载入系统。

可以启动Erlang shell然后输入命令`code:get_path()`来找到当前的载入路径值。这里有一个例子：

```
1> code:get_path().
[".",
 "/usr/local/lib/erlang/lib/kernel-2.15/ebin",
 "/usr/local/lib/erlang/lib/stdlib-1.18/ebin",
 "/home/joe/installed/proper/ebin",
 "/usr/local/lib/erlang/lib/xmerl-1.3/ebin",
 "/usr/local/lib/erlang/lib/wx-0.99.1/ebin",
 "/usr/local/lib/erlang/lib/webtool-0.8.9.1/ebin",
 "/usr/local/lib/erlang/lib/typer-0.9.3/ebin",
 "/usr/local/lib/erlang/lib/tv-2.1.4.8/ebin",
 "/usr/local/lib/erlang/lib/tools-2.6.6.6/ebin",
...]
```

以下是两个最常用来操作载入路径的函数：

- -spec code:add_patha(Dir) => true | {error, bad_directory}
  向载入路径的开头添加一个新目录Dir。
- -spec code:add_pathz(Dir) => true | {error, bad_directory}
  向载入路径的末端添加一个新目录Dir。

使用哪个通常无关紧要。唯一要留心的是add_patha和add_pathz是否会导致不同的结果。如果你怀疑载入了错误的模块，可以调用code:all_loaded()（返回一个已加载模块的总列表）或code:clash()来帮助调查哪里出了错。

code模块里的其他一些函数也能够操作路径，不过你很可能用不到这些函数，除非正在做一些古怪的系统编程。

通常的惯例是把这些命令放在主目录（home directory）里一个名为.erlang的文件内。

也可以用这样的命令来启动Erlang：

```
$ erl -pa Dir1 -pa Dir2 ... -pz DirK1 -pz DirK2
```

-pa Dir标识会把Dir添加到代码搜索路径的开头，-pz Dir则会把此目录添加到代码路径的末端。 10

10.1.2 在系统启动时执行一组命令

我们已经看到了该如何在主目录的.erlang文件里设置载入路径。事实上，你可以把任意的Erlang代码放入这个文件。启动Erlang时，它会首先读取并执行此文件里的所有命令。

假设.erlang文件内容如下：

```
io:format("Hi, I'm in your .erlang file~n").
...
```

启动系统时，我们会看到以下输出：

```
$ erl
...
```

```
Hi, I'm in your .erlang file
Eshell V5.9  (abort with ^G)
1>
```

如果Erlang启动时的当前目录里已经有一个.erlang文件，它就会优先于主目录里的.erlang。通过这种方式，可以让Erlang根据不同的启动位置表现出不同的行为。这对特定的应用程序来说可能很有用。在这种情况下，多半应该把一些打印语句添加到启动文件里，否则你可能会忘记本地启动文件的存在，从而感到非常困惑。

提示　在某些系统里，主目录的位置并不清晰，或者可能和你认为的不一样。要找出Erlang认定的主目录位置，可以这么做：

```
1> init:get_argument(home).
{ok,[["/home/joe"]]}
```

通过上面的信息可以推断Erlang认为你的主目录是/home/joe。

10.2　运行程序的不同方式

Erlang程序被保存在模块里。编写完程序之后，必须编译它才能运行。也可以通过无需编译的escript来直接运行程序。

接下来的几节将展示如何用多种方式编译和运行两个程序。这两个程序略有不同，启动和停止它们的方式也有区别。

第一个程序hello.erl只打印“Hello world”。它不负责启动和停止系统，也不需要访问命令行参数。与之相对，第二个程序fac需要访问命令行参数。

下面是基本的程序。它打印出的字符串是“Hello world”再加一个换行符（~n在Erlang的io和io_lib模块里被解释为换行符）。

hello.erl

```
-module(hello).
-export([start/0]).

start() ->
    io:format("Hello world~n").
```

可用三种方式编译和运行它。

10.2.1　在 Erlang shell 里编译和运行

我们从启动Erlang shell开始。

```
$ erl
...
1> c(hello).
{ok,hello}
```

```
2> hello:start().
Hello world
ok
```

快速脚本编程

我们经常希望能在操作系统的命令行里执行任意的Erlang函数。-eval参数非常适合进行快速脚本编程。

这里有一个例子：

```
erl -eval 'io:format("Memory: ~p~n", [erlang:memory(total)]).'\
    -noshell -s init stop
```

10.2.2 在命令提示符界面里编译和运行

可以直接在命令提示符界面里编译程序。如果只是想编译一些代码而不运行它们，这就是最简单的方式。它的做法如下：

```
$ erlc hello.erl
$ erl -noshell -s hello start -s init stop
Hello world
$
```

注意 本章所有的shell命令都假定用户已经在他们的系统里安装了合适的shell，并且erl和erlc等命令能直接在shell里执行。具体如何配置系统会随系统和时间点的不同而有所区别。你可以在Erlang网站[①]和主开发存档[②]里找到最新的资料。

第1行的erlc hello.erl编译了hello.erl文件，生成了一个名为hello.beam的目标代码文件。第二个命令有三个选项。

- -noshell以不带交互式shell的方式启动Erlang（因此不会看到Erlang的“徽标”，也就是通常系统启动时首先显示的那些信息）。
- -s hello start运行hello:start()函数。注意：使用-s Mod ...选项时，Mod必须是已编译的。
- -s init stop在之前的命令完成后执行init:stop()函数，从而停止系统。

erl -noshell ...命令可以放在shell脚本里，所以通常会制作一个shell脚本来运行程序，里面会设置路径（用-pa Directory）并启动程序。

在这个例子里用了两个-s ..命令。命令行里的函数数量是不受限制的。每个-s ...命令都由一个apply语句执行，运行完毕后再执行下一个命令。

10

① http://www.erlang.org

② https://github.com/erlang/otp

这里有一个启动hello.erl的例子：

hello.sh

```
#!/bin/sh
erl -noshell -pa /home/joe/2012/book/JAERLANG/Book/code\
            -s hello start -s init stop
```

注意　这个脚本需要一个指向hello.beam文件所在目录的绝对路径。所以虽然这个脚本能在我的机器上工作，但是你必须修改一下才能让它在你的机器上运行。

为了运行shell脚本，我们需要chmod这个文件（只需一次），然后就可以运行它了。

```
$ chmod u+x hello.sh
$ ./hello.sh
Hello world
```

10.2.3　作为Escript运行

可以用escript来让程序直接作为脚本运行，无需事先编译它们。为了让hello作为escript运行，我们创建了下面的文件：

hello

```
#!/usr/bin/env escript

main(Args) ->
    io:format("Hello world~n").
```

这个文件必须包含一个main(Args)函数。当它从操作系统的shell里调用时，Args会包含一列用原子表示的命令行参数。在Unix系统上，无需编译就可以立即运行，就像下面这样：

```
$ chmod u+x hello
$ ./hello
Hello world
```

注意　这个文件的文件模式必须设置为“可执行”（Unix系统上可以输入命令chmod u+x File），这个操作只须做一次，不是每次运行程序时都要做。

在开发过程中导出函数

编写代码时，一件很麻烦的事就是要不断地添加和移除程序的导出声明，从而使导出的程序可以在shell里运行。

-compile(export_all).这个特殊声明能让编译器导出模块里的每一个函数。这么做能让你在编写代码时轻松很多。

完成代码开发后，应该注释掉export_all声明并添加合适的导出声明。这么做的原因有两个。首先，等你稍后回来阅读代码时，就会知道导出的那些函数才是重要的。其他所有函数都不能在模块之外调用，所以可以用任何你喜欢的方式修改它们，只要确保对导出函数的接口保持不变即可。其次，如果编译器准确知道模块导出了哪些函数，就能生成更好的代码。

请注意，使用-compile(export_all).会大大增加dialyzer分析代码的难度。

10.2.4 带命令行参数的程序

“Hello world”没有任何参数。让我们用一个计算阶乘的程序来重复这个练习。它接受单个参数。

首先是它的代码：

fac.erl

```
-module(fac).
-export([fac/1]).

fac(0) -> 1;
fac(N) -> N*fac(N-1).
```

可以像这样编译fac.erl并在Erlang shell里运行它：

```
$ erl
...
1> c(fac).
{ok,fac}
2> fac:fac(25).
15511210043330985984000000
```

如果希望在命令行里运行这个程序，就需要修改它，让它能够接受命令行参数。

fac1.erl

```
-module(fac1).
-export([main/1]).

main([A]) ->
    I = list_to_integer(atom_to_list(A)),
    F = fac(I),
    io:format("factorial ~w = ~w~n",[I, F]),
    init:stop().

fac(0) -> 1;
fac(N) -> N*fac(N-1).
```

然后可以编译并运行它。

```
$ erlc fac1.erl
$ erl -noshell -s fac1 main 25
factorial 25 = 15511210043330985984000000
```

注意　这个函数的名称main没有什么特殊含义，你可以给它取任何名字。重点是函数名和命令行里的名称要一致。

最后，可以把它作为escript运行。

factorial

```
#!/usr/bin/env escript
main([A]) ->
    I = list_to_integer(A),
    F = fac(I),
    io:format("factorial ~w = ~w~n",[I, F]).

fac(0) -> 1;
fac(N) ->
    N * fac(N-1).
```

无需编译就可以运行它，如下所示：

```
$ ./factorial 25
factorial 25 = 15511210043330985984000000
```

10.3　用 makefile 使编译自动化

当我编写一个很大的程序时，我喜欢让它尽可能地自动化。这么做有两个原因。首先，从长远来看这能减少输入操作：一遍遍输入相同的命令来测试和重复测试程序需要大量的按键操作，我可不想磨损我的手指头。

其次，我经常会暂停手头上的事，转而处理别的某个项目，再回到某个暂停的项目可能会是几个月以后的事情了。当我回来时经常已经忘了如何构建项目里的代码，这时候make就能派上大用场了！

make是我唯一的任务自动化工具，我用它编译和分发Erlang代码。我的大多数makefile[①]都极其简单，而且我还有一个简单的模板，它能满足我的大多数需要。

我不会对makefile做总体的介绍，而是展示我认为对编译Erlang程序有用的那种形式。我们会特别介绍本书所附的makefile，这样你就能理解它们并制作自己的makefile了。

一个makefile模板

下面这个模板是我制作大多数makefile的基础：

① http://en.wikipedia.org/wiki/Make

Makefile.template

```
# 别碰这几行
.SUFFIXES: .erl .beam .yrl

.erl.beam:
        erlc -W $<
.yrl.erl:
        erlc -W $<

ERL = erl -boot start_clean

# 这里是一个想要编译的Erlang模块列表。
# 如果这些模块在一行里放不下，
# 就在行尾添加一个 \ 字符然后在下一行继续。

# 编辑下面这几行
MODS = module1 module2 \
       module3 ... special1 ...\
       ...
       moduleN

# 任何makefile里的第一个目标就是默认的目标。
# 如果只输入了"make"，系统就会假定为"make all"。
#（因为"all"是这个makefile里的第一个目标）

all: compile

compile: ${MODS:%=%.beam} subdirs

## 此处添加特殊的编译要求

special1.beam: special1.erl
        ${ERL} -Dflag1 -W0 special1.erl

## 从makefile里运行应用程序

application1: compile
        ${ERL} -pa Dir1  -s application1 start Arg1 Arg2

# subdir目标会编译子目录里的代码
# sub-directories

subdirs:
        cd dir1; $(MAKE)
        cd dir2; $(MAKE)
        ...

# 移除所有代码
```

```
clean:
        rm -rf *.beam erl_crash.dump
        cd dir1; $(MAKE) clean
        cd dir2; $(MAKE) clean
```

这个makefile的开头部分有一些规则，用于编译Erlang模块和扩展名为`.yrl`的文件（这些文件包含了解析器定义，供Erlang解析器生成程序使用）。Erlang的解析器生成程序名为`yecc`（它是Erlang版的`yacc`，后者是yet another compiler compiler的缩写，即“又一个编译器的编译器”，详情请见在线教程[1]）。

重要的部分是下面这一行：

```
MODS = module1 module2
```

它是一个清单，包含了我想要编译的所有Erlang模块。

`MODS`清单里的任何模块都会用Erlang命令`erlc Mod.erl`（`Mod`是模块名）编译。有些模块可能需要特殊对待（例如模板文件里的`special1`模块），所以里面有一条单独的规则对此进行处理。

makefile里有一些目标（targets）。目标是一个由字母数字组成的字符串，从行首开始，到冒号（`:`）结束。在这个makefile模板里，`all`、`compile`和`special1.beam`都是目标。要运行这个makefile，你需要输入shell命令。

```
$ make [Target]
```

参数`Target`是可选的。如果省略了`Target`，系统就假定是文件里的第一个目标。在前面的例子中，如果没有在命令行里指定目标，`all`就成了假定的目标。

如果我想要编译所有的软件并运行`application1`，就会输入命令`make application1`。如果我想让它变成默认的行为（只需输入命令`make`就能实现），就会移动定义目标`application1`的那几行，让它们成为makefile里的第一个目标。

目标`clean`会移除所有已编译的Erlang对象代码文件和`erl_crash.dump`文件。这个故障转储（crash dump）文件所包含的信息有助于对程序进行调试。详情请见10.4.5节。

精简makefile模板

我不喜欢杂乱的软件，所以经常一开始就会移除makefile模板里那些与应用程序无关的行。这能让makefile变得更简短易读。也可以制作一个公用的makefile，把它包含在所有的makefile里，并将这些makefile里的变量作为参数。

这个过程完成后，我就有了一个高度精简的makefile，就像下面这个：

```
.SUFFIXES: .erl .beam

.erl.beam:
    erlc -W $<
```

---

① http://erlang.org/doc/man/yecc.html

```
ERL = erl -boot start_clean

MODS = module1 module2 module3

all: compile
	${ERL} -pa '/home/joe/.../this/dir' -s module1 start

compile: ${MODS:%=%.beam}

clean:
	rm -rf *.beam erl_crash.dump
```

10.4 当坏事发生

这一节列出了一些常见问题，以及它们的解决方法。

10.4.1 停止 Erlang

有时候Erlang会根本停不下来，以下是一些可能的原因。

- shell没有反应。
- Ctrl+C操作被禁用了。
- Erlang启动时设置了`-detached`标识，所以你可能不会注意到它正在运行。
- Erlang启动时设置了`-heart Cmd`选项。这个选项会建立一个操作系统监控进程来监控Erlang的操作系统进程。如果Erlang的操作系统进程崩溃了，`Cmd`就会被执行。通常`Cmd`只是简单地重启Erlang系统。这是我们制作容错式节点时使用的技巧之一——如果Erlang自身崩溃了（这理应不该出现），就会自动重启。解决方法是找到心跳（heartbeat）进程（用类Unix系统的`ps`和Windows的任务管理器），先终止它，然后再终止Erlang进程。
- 某处可能出了严重的问题，遗留了一个失去联系的僵尸Erlang进程。

10

10.4.2 未定义（缺失）的代码

如果代码载入器找不到你试图运行的代码所在的模块（因为代码搜索路径不正确），就会遇到一条`undef`错误消息。这里有一个例子：

```
1> glurk:oops(1,23).
** exception error: undefined function glurk:oops/2
```

实际上，名为`glurk`的模块并不存在，但这里的问题不在于此。应该关注的是这段错误消息。它告诉我们系统尝试调用`glurk`模块里带有两个参数的函数`oops`。因此，有四种可能的情况。

- `glurk`模块确实不存在——毫无踪迹，无处可寻。这多半是因为拼写错误。
- `glurk`模块存在，但未被编译。系统一直在代码搜索路径里找的是一个名为`glurk.beam`的文件。

- glurk模块存在且已编译，但glurk.beam所处的目录不属于代码搜索路径。要修复这个问题，必须更改搜索路径。
- 代码载入路径里有多个不同版本的glurk，而我们选择了一个错误的版本。这种错误很罕见，但也是有可能发生的。

如果怀疑发生了这种错误，可以运行code:clash()函数，它会报告代码搜索路径里所有重复的模块。

有谁看到我的分号了？

如果忘记在函数的子句之间放置分号，或者用句号代替，那就有麻烦了，而且是大麻烦。

如果在模块bar的第1234行定义了一个foo/2函数，并且把句号放在了分号的位置上，编译器就会提示：

bar.erl:1234里的函数foo/2已被定义过。

所以，别这么做。请确保你的子句总是用分号隔开。

10.4.3　shell没有反应

如果shell不再响应命令，那么可能发生了几种状况。shell进程自身可能崩溃了，或者你给出了一个永远不会终止的命令。甚至可能只是忘记输入一个关闭引号，或者忘记在命令结尾输入句号加回车。

无论是什么原因，都可以按下Ctrl+G来中断当前的shell，然后进行如下处理：

```
❶ 1> receive foo -> true end.
   ^G
   User switch command
❷ --> h
   c [nn]   - connect to job
   i [nn]   - interrupt job
   k [nn]   - kill job
   j        - list all jobs
   s        - start local shell
   r [node] - start remote shell
   q        - quit erlang
   ? | h    - this message
❸ --> j
   1* {shell,start,[init]}
❹ --> s
   --> j
   1  {shell,start,[init]}
   2* {shell,start,[]}
❺ --> c 2
   Eshell V5.5.1  (abort with ^G)
   1> init:stop().
   ok
   2> $
```

❶ 此处我们让shell接收一个foo消息。但因为没人向shell发送这个消息，所以shell会处于永久等待的状态。我们按下Ctrl+G后进入shell。

❷ 系统进入“shell JCL”（Job Control Language，任务控制语言）模式。我们输入h来获得一些帮助。

❸ 输入j列出了所有的任务。任务1带有一个星号标记，意思是默认shell。所有带有可选参数[nn]的命令都使用默认shell，除非提供了特定的参数。

❹ 输入命令s启动了一个新的shell，然后再输入j。这一次我们可以看到两个shell，分别标记为1和2，shell 2现在成了默认shell。

❺ 我们输入c 2，这样就连上了新启动的shell 2。然后我们停止了系统。

如你所见，我们可以让多个shell同时运行，通过按Ctrl+G并输入合适的命令来切换它们。我们甚至还可以用r命令在远程节点上启动一个shell。

10.4.4 我的makefile不工作

makefile能出什么问题？事实上有不少。但本书不是关于makefile的，所以我只会讨论最常见的错误。下面这两个错误是我犯得最多的。

- ❑ makefile里的空格。makefile极其挑剔，虽然肉眼看不见，但是makefile里每个缩进行都应该以一个制表符开头（除了连续行，也就是它的前一行以一个\字符结尾）。如果此处有空格，make就会被干扰，导致错误出现。
- ❑ Erlang文件缺失。如果MODS里声明的某一个模块不存在，就会得到错误消息。举个例子，假设MODS包含一个名为glurk的模块，但代码目录里不存在名为glurk.erl的文件。在这种情况下，make就会出错并给出下面的消息：

```
$ make
make: *** No rule to make target `glurk.beam',
          needed by `compile'.  Stop.
```

另一种情况是模块存在，但makefile里的模块名有拼写错误。

10.4.5 Erlang崩溃而你想阅读故障转储文件

如果Erlang崩溃了，它会留下一个名为erl_crash.dump的文件。这个文件的内容也许能提示你问题出在哪里。有一个基于Web的故障分析器可以用来分析故障转储文件。要启动这个分析器，请输入以下命令：

```
1> crashdump_viewer:start().
WebTool is available at http://localhost:8888/
Or  http://127.0.0.1:8888/
ok
```

然后把浏览器指向http://localhost:8888/，这样就可以愉快地浏览错误日志了。

10.5　获取帮助

在Unix系统里，可以用下面的方式访问手册页：

```
$ erl -man erl
NAME
erl - The Erlang Emulator

DESCRIPTION
The erl program starts the Erlang runtime system.
The exact details (e.g. whether erl is a script
or a program and which other programs it calls) are system-dependent.
...
```

还可以用下面的方式获取各个模块的帮助信息：

```
$ erl -man lists
MODULE
lists - List Processing Functions

DESCRIPTION
This  module  contains  functions  for  list  processing.
The functions are organized in two groups:
...
```

注意　Unix系统默认是不安装手册页的。如果命令`erl -man ...`无效，就需要安装手册页。所有的手册页都存放在一个单独的压缩包里[①]。这些手册页应当被解压到Erlang的安装根目录中（通常是`/usr/local/lib/erlang`）。

还有一组HTML文件形式的文档可供下载。这些HTML文档在Windows下是默认安装的，可以通过开始菜单里的Erlang条目访问。

10.6　调节运行环境

Erlang shell有许多内置命令。可以通过shell命令`help()`进行查看。

```
1> help().
** shell internal commands **
b()        -- display all variable bindings
e(N)       -- repeat the expression in query <N>
f()        -- forget all variable bindings
f(X)       -- forget the binding of variable X
h()        -- history
...
```

① http://www.erlang.org/download.html

所有这些命令都定义在shell_default模块里。

如果想定义自己的命令，只需创建一个名为user_default的模块即可。这里有一个例子：

user_default.erl

```
-module(user_default).

-compile(export_all).

hello() ->
    "Hello Joe how are you?".

away(Time) ->
    io:format("Joe is away and will be back in ~w minutes~n",
              [Time]).
```

当编译它并将它放置在载入路径的某个地方后，就可以不带模块名调用user_default里的任意函数了。

```
1> hello().
"Hello Joe how are you?"
2> away(10).
Joe is away and will be back in 10 minutes
ok
```

掌握了详细的具体知识后，可以来看看并发程序了。这才是真正有意思的部分。

10.7　练习

创建一个新目录，然后把本章的makefile模板复制进去。编写一个Erlang小程序并把它保存在此目录中。给makefile和Erlang代码添加一些命令，让它们在你在输入make后能自动运行一组单元测试（参见4.1.3节）。

Part 3 第三部分

并发和分布式程序

这一部分将介绍并发和分布式 Erlang。以顺序 Erlang 为基础，你将学会如何编写并发程序，并在分布式的计算机网络中运行它们。

第 11 章

现实世界中的并发

让我们暂时忘却编程，思考一下现实世界里发生着什么。



● *我们理解并发。*

我们的大脑天生就对并发有着深刻的理解。大脑里一个名为杏仁核的区域让我们能迅速对刺激作出反应。如果没有这种反应，我们就会死亡。意识思维实在是太慢了，当“踩下刹车”这个念头形成时，我们已经这么做了。

当我们驾车行驶在主干道上，脑子里会时刻定位着数十甚至数百辆车。这是在没有意识思维参与的情况下做到的。如果我们做不到这一点，多半就没命了。

● *世界是并行的。*

如果我们想让编写的程序有着现实世界里其他对象的行为，这些程序就会是并发架构的。

这就是我们应该用并发编程语言来编写程序的原因。

然而，我们经常用顺序编程语言来编写现实世界里的应用程序。这样做会带来不必要的困难。

如果使用一种为编写并发应用程序而设计的语言，进行并发开发就会简单得多。

- Erlang程序反映了我们思考和交流和方式。

我们没有共享“内存”（也就是记忆）。我有我的记忆，你有你的记忆。我们各有一个大脑，它们并不相连。为了改变你的记忆，我会向你发送一个消息：通过说话，或者挥舞手臂。

你倾听，观察，然后改变了记忆。但是，如果不问你问题或者观察你的反应，我就无法知道你是否收到了我的消息。

这就是Erlang进程的工作方式。Erlang进程没有共享内存，每个进程都有它自己的内存。要改变其他某个进程的内存，必须向它发送一个消息，并祈祷它能收到并理解这个消息。

要确定另一进程收到了你的消息并改变了它的内存，就必须询问它（通过向它发送一条消息）。这就是我们的交流方式。

苏：嗨，比尔，我的电话号码是345-678-1234。

苏：你听到了吗？

比尔：是的，你的电话是345-678-1234。

这些交流模式都是你我所熟知的。我们从出生起就逐渐学会了如何与世界交互，就是观察它，向它发送消息并观察回应。

- 人类表现为独立的个体，通过发送消息进行交流。

这就是Erlang进程的工作方式，也是我们的工作方式，因此要理解Erlang程序很容易。

一个Erlang程序会包含几十、几千、甚至几十万个小进程。所有这些进程都是独立运作的。它们通过发送消息来相互交流。每个进程都拥有一块私有内存区域。它们表现得就像是一大群人在一个巨大的房间里喋喋不休。

这使得Erlang程序天生易于管理和扩展。假设我们有10个人（进程），而他们有太多的工作要做，我们可以怎么办？找更多的人过来。我们要如何管理这群人？很简单——大声把命令告诉他们（广播）就可以了。

Erlang进程不共享内存，因此使用内存时无需加锁。有锁的地方就会有钥匙，而钥匙是容易丢失的。当你丢了钥匙会发生什么？你会惊慌得不知所措。当你在软件系统里丢了钥匙，使锁出现问题时也会如此。

分布式软件系统里只要有锁和钥匙，就总会出问题。

Erlang没有锁，也没有钥匙。

- 如果有人死亡，其他人会注意到。

如果我在一个房间里突然倒下死去，很可能就会有人注意到（好吧，至少我希望如此）。Erlang进程就像人类一样，有时会死去。但和人类不同的是，当它们死亡时，会用尽最后一口气喊出导致它们死亡的准确原因。

想象一个挤满人的房间里突然有一个人倒下死去。就在那一刻，他说“我的心脏病发作了”或者“我吃得太多，胃爆炸了”。Erlang进程就是这么做的。一个进程可能会在临死时说：“我是因为有人要求我除以零而死的。”另一个可能会说：“我是因为有人问我空列表的最后一个元素是

什么而死的。”

现在，在这个挤满人的房间里，我们可以设想有些人被特别指派从事清理尸体的工作。让我们假设有简和约翰两个人。如果简死了，约翰会处理一切与简的死亡有关的问题。如果约翰死了，简会处理这些问题。简和约翰通过一种不可见的约定连接在一起，这个约定是如果其中一人死亡，另一人就会处理一切由此产生的问题。

Erlang的错误检测正是使用的这种方式。进程可以相互连接。如果其中一个进程挂了，另一个进程就会得到一个说明前者死亡原因的错误消息。

大致就是这么一回事。

Erlang程序就是这么工作的。

到目前为止，我们学到的以下内容。

- Erlang程序由大量进程组成。这些进程间能相互发送消息。
- 这些消息也许能被其他进程收到和理解，也许不能。如果想知道某个消息是否已被对方进程收到和理解，就必须向该进程发送一个消息并等待回复。
- 进程可以成对相互连接。如果某一对互连进程的其中一个挂了，另一个进程就会收到一个说明前者死亡原因的消息。

这个简单的编程模型是一个大模型的一部分，我把这个大模型称为面向并发编程（concurrency-oriented programming）。

在下一章里，我们将开始编写并发程序。我们需要学习三个新的基本函数：`spawn`、`send`（使用`!`操作符）和`receive`。然后就能编写一些简单的并发程序了。

如果一个进程挂了，另一个进程（如果与前者相连的话）就会注意到。这是第13章的主题。

在阅读后面这两章的过程中，可以联想一下房间里的人。人就是进程。房间里的人都有他们的私人记忆，进程也是如此。要改变你的记忆，就需要我说给你听。这就是发送和接收消息。我们有了小孩，这就是分裂（spawn）。我们死了，就是进程退出。

第 12 章

并发编程

了解顺序Erlang后，编写并发程序就很简单了。只需要三个新的基本函数：`spawn`、`send`和`receive`。`spawn`创建一个并行进程，`send`向某个进程发送消息，`receive`则是接收消息。

Erlang的并发是基于进程（process）的。进程是一些独立的小型虚拟机，可以执行Erlang函数。

你肯定曾经接触过进程，但仅仅是在操作系统的上下文环境里。在Erlang里，进程隶属于编程语言，而非操作系统。这就意味着Erlang的进程在任何操作系统上都会具有相同的逻辑行为，这样，就能编写可移植的并发代码，让它在任何支持Erlang的操作系统上运行。

在Erlang里：

- 创建和销毁进程是非常快速的；
- 在进程间发送消息是非常快速的；
- 进程在所有操作系统上都具有相同的行为方式；
- 可以拥有大量进程；
- 进程不共享任何内存，是完全独立的；
- 进程唯一的交互方式就是消息传递。

出于这些原因，Erlang有时会被称为是一种纯消息传递式语言。

如果你没有进程编程的经验，可能听说过它很有难度的传言。你多半听过一些恐怖故事，涉及内存冲突、竞争状况、共享内存破坏，等等。但在Erlang里，进程编程是很简单的。

12.1 基本并发函数

我们在顺序编程里学到的知识同样适用于并发编程。要做的只是加上下面这几个基本函数。

- Pid = spawn(Mod, Func, Args)

  创建一个新的并发进程来执行`apply(Mod, Func, Args)`。这个新进程和调用进程并列运行。`spawn`返回一个`Pid`（process identifier的简称，即进程标识符）。可以用`Pid`来给此进程发送消息。请注意，元数为`length(Args)`的`Func`函数必须从`Mod`模块导出。

  当一个新进程被创建后，会使用最新版的代码定义模块。

- Pid = spawn(Fun)

  创建一个新的并发进程来执行`Fun()`。这种形式的`spawn`总是使用被执行fun的当前值，而且这个fun无需从模块里导出。

  这两种`spawn`形式的本质区别与动态代码升级有关。12.8节会讨论如何从这两种`spawn`形式中做出选择。

- Pid ! Message

  向标识符为`Pid`的进程发送消息`Message`。消息发送是异步的。发送方并不等待，而是会继续之前的工作。`!`被称为发送操作符。

  `Pid ! M`被定义为`M`。因此，`Pid1 ! Pid2 !...! Msg`的意思是把消息`Msg`发送给`Pid1`、`Pid2`等所有进程。

- receive ... end

  接收发送给某个进程的消息。它的语法如下：

```
receive
   Pattern1 [when Guard1] ->
       Expressions1;
   Pattern2 [when Guard2] ->
       Expressions2;
   ...
end
```

  当某个消息到达进程后，系统会尝试将它与`Pattern1`（以及可选的关卡`Guard1`）匹配，如果成功就执行`Expressions1`。如果第一个模式不匹配，就会尝试`Pattern2`，以此类推。如果没有匹配的模式，消息就会被保存起来供以后处理，进程则会开始等待下一条消息。12.5节中会介绍更多的细节。

  接收语句里的模式和关卡和我们定义函数时使用的模式和关卡具有相同的语法形式和含义。

好了，就是这些。不需要线程、锁、信号和人工控制。

到目前为止，我们粗略介绍了`spawn`、`send`和`receive`的工作方式。当`spawn`命令被执行时，系统会创建一个新的进程。每个进程都带有一个邮箱，这个邮箱是和进程同步创建的。

给某个进程发送消息后，消息会被放入该进程的邮箱。只有当程序执行一条接收语句时才会读取邮箱。

通过这三个基本函数，我们可以把4.1节里的`area/1`函数转变为一个进程。提醒一下，定义`area/1`函数的代码如下：

geometry.erl

```
area({rectangle, Width, Height}) -> Width * Height;
area({square, Side})             -> Side * Side.
```

现在把这个函数改写成一个进程。为此，我们从area函数的参数里取了两个模式，然后把它们重置为接收语句里的模式。

area_server0.erl

```
-module(area_server0).
-export([loop/0]).

loop() ->
    receive
        {rectangle, Width, Ht} ->
            io:format("Area of rectangle is ~p~n",[Width * Ht]),
            loop();
        {square, Side} ->
            io:format("Area of square is ~p~n", [Side * Side]),
            loop()
    end.
```

可以在shell里创建一个执行loop/0的进程。

```
1> Pid = spawn(area_server0, loop, []).
<0.36.0>
2> Pid ! {rectangle, 6, 10}.
Area of rectangle is 60
{rectangle,6,10}
3> Pid ! {square, 12}.
Area of square is 144
{square, 144}
```

我们在第1行里创建了一个新的并行进程。spawn(area_server, loop, [])会创建一个执行area_server:loop()的并行进程，然后返回Pid，也就是打印出来的<0.36.0>。

在第2行里向这个进程发送了一个消息。这个消息匹配loop/0接收语句里的第一个模式：

```
loop() ->
    receive
        {rectangle, Width, Ht} ->
            io:format("Area of rectangle is ~p~n",[Width * Ht]),
            loop()
        ...
```

收到消息之后，这个进程打印出矩形的面积。最后，shell打印出{rectangle, 6, 10}，这是因为Pid ! Msg的值被定义为Msg。

12

12.2　客户端-服务器介绍

客户端-服务器架构是Erlang的中心。传统的客户端-服务器架构是指一个分隔客户端与服务器的网络。大多数情况下客户端会有多个实例，而服务器只有一个。服务器这个词经常会让人联想到专业机器上运行重量级软件的画面。

我们的实现机制则要轻量得多。客户端–服务器架构里的客户端和服务器是不同的进程，它们之间的通信使用普通的Erlang消息传递机制。客户端和服务器可以运行在同一台机器上，也可以运行在不同的机器上。

*客户端*和*服务器*这两个词是指这两种进程所扮演的角色：客户端总是通过向服务器发送一个*请求*来发起计算。服务器计算后生成回复，然后发送一个*响应*给客户端。

下面来编写我们的第一个客户端–服务器应用程序。首先，对上一节里编写的程序做一些小小的修改。

在上一个程序里，我们只需要向某个进程发送请求，然后接收它并打印出来。现在要做的是向发送原请求的进程发送一个响应。问题是，我们不知道该把响应发给谁。要发送一个响应，客户端必须加入一个服务器可以回复的地址。这就像是给某人写信——如果你想得到回复，最好把你的地址写在信中！

因此，发送方必须加入一个回复地址。要做到这一点，可以把：

```
Pid ! {rectangle, 6, 10}
```

修改成下面这样：

```
Pid ! {self(),{rectangle, 6, 10}}
```

`self()`是客户端进程的标识符。

为了响应请求，我们必须把接收请求的代码从：

```
loop() ->
    receive
        {rectangle, Width, Ht} ->
            io:format("Area of rectangle is ~p~n",[Width * Ht]),
            loop()
        ...
```

修改成下面这样：

```
loop() ->
    receive
        {From, {rectangle, Width, Ht}} ->
            From ! Width * Ht,
            loop();
        ...
```

请注意我们是如何把计算结果发回由`From`参数指定的进程的。因为客户端把这个参数设置成它自己的ID，所以能收到结果。

发送请求的进程通常称为*客户端*。接收请求并回复客户端的进程称为*服务器*。

另外，最佳实践是确认发送给进程的每一个消息都已收到。如果发送给进程的消息不匹配原始接收语句里的任何一个模式，这条消息就会遗留在进程邮箱里，永远无法接收。为了解决这个问题，我们在接收语句的最后加了一个子句，让它能匹配所有发送给此进程的消息。

最后，添加一个名为`rpc`（remote procedure call的缩写，即远程过程调用）的实用小函数，

它封装了向服务器发送请求和等待响应的代码。

area_server1.erl

```
rpc(Pid, Request) ->
    Pid ! {self(), Request},
    receive
        Response ->
            Response
    end.
```

把所有这些合并到一起，得到了下面的代码：

area_server1.erl

```
-module(area_server1).
-export([loop/0, rpc/2]).
rpc(Pid, Request) ->
    Pid ! {self(), Request},
    receive
        Response ->
            Response
    end.
loop() ->
    receive
        {From, {rectangle, Width, Ht}} ->
            From ! Width * Ht,
            loop();
        {From, {circle, R}} ->
            From !  3.14159 * R * R,
            loop();
        {From, Other} ->
            From ! {error,Other},
            loop()
    end.
```

可以在shell里试验一下它。

```
1> Pid = spawn(area_server1, loop, []).
<0.36.0>
2> area_server1:rpc(Pid, {rectangle,6,8}).
48
3> area_server1:rpc(Pid, {circle,6}).
113.097
4> area_server1:rpc(Pid, socks).
{error,socks}
```


这段代码有个小问题。在`rpc/2`函数里，我们向服务器发送请求然后等待响应。但我们并不是等待来自服务器的响应，而是在等待任意消息。如果其他某个进程在客户端等待来自服务器的消息时向它发送了一个消息，客户端就会将此消息错误解读为来自服务器的响应。要纠正这个问题，可以把接收语句的形式修改如下：

```
loop() ->
    receive
        {From, ...} ->
            From ! {self(), ...}
            loop()
        ...
    end.
```

再把rpc修改如下：

```
rpc(Pid, Request) ->
    Pid ! {self(), Request},
    receive
        {Pid, Response} ->
            Response
    end.
```

调用rpc函数时，Pid会被绑定为某个值，因此{Pid, Response}这个模式里的Pid已绑定，而Response未绑定。这个模式只会匹配包含一个双元素元组（第一个元素是Pid）的消息。所有别的消息都会进入队列。（receive提供了选择性接收的功能，我会在后面介绍。）修改之后的代码如下：

area_server2.erl

```
-module(area_server2).
-export([loop/0, rpc/2]).
rpc(Pid, Request) ->
    Pid ! {self(), Request},
    receive
        {Pid, Response} ->
            Response
    end.
loop() ->
    receive
        {From, {rectangle, Width, Ht}} ->
            From ! {self(), Width * Ht},
            loop();
        {From, {circle, R}} ->
            From !  {self(), 3.14159 * R * R},
            loop();
        {From, Other} ->
            From ! {self(), {error,Other}},
            loop()
    end.
```

它的工作方式和预期的一致。

```
1> Pid = spawn(area_server2, loop, []).
<0.37.0>
2> area_server2:rpc(Pid, {circle, 5}).
78.5397
```

还有最后一点可改进的地方。我们可以让spawn和rpc隐藏在模块内。请注意，还需要把spawn的参数（也就是loop/0）从模块中导出。这是一种好的做法，因为它能让我们在不改变客户端代码的情况下修改服务器的内部细节。最终的代码如下：

area_server_final.erl

```
-module(area_server_final).
-export([start/0, area/2, loop/0]).

start() -> spawn(area_server_final, loop, []).

area(Pid, What) ->
    rpc(Pid, What).
rpc(Pid, Request) ->
    Pid ! {self(), Request},
    receive
        {Pid, Response} ->
            Response
    end.
loop() ->
    receive
        {From, {rectangle, Width, Ht}} ->
            From ! {self(), Width * Ht},
            loop();
        {From, {circle, R}} ->
            From !  {self(), 3.14159 * R * R},
            loop();
        {From, Other} ->
            From ! {self(), {error,Other}},
            loop()
    end.
```

我们调用函数start/0和area/2（之前称为spawn和rpc）来运行它。这些新名称更好一些，因为它们能更准确地描述服务器的行为。

```
1> Pid = area_server_final:start().
<0.36.0>
2> area_server_final:area(Pid, {rectangle, 10, 8}).
80
3> area_server_final:area(Pid, {circle, 4}).
50.2654
```

这样就完成了一个简单的客户端–服务器模块。所需要的就是三个基本函数：spawn、send和receive。这种模式会以各类变种的形式不断重复出现，变化虽然可大可小，但基本的概念是不变的。

12

12.3　进程很轻巧

在这个阶段，你可能会担心性能问题。毕竟，如果创建数百或者数千个Erlang进程，就必须付出一定的代价。让我们来看看代价有多大。

我们将执行一些分裂操作，创建大量的进程，并计算要花费多长时间。下面是一个程序，请注意在这里用的是spawn(Fun)，并且被分裂出的函数并不需要从模块里导出：

processes.erl

```
-module(processes).

-export([max/1]).

%% max(N)

%% 创建N个进程然后销毁它们
%% 看看需要花费多长时间

max(N) ->
    Max = erlang:system_info(process_limit),
    io:format("Maximum allowed processes:~p~n",[Max]),
    statistics(runtime),
    statistics(wall_clock),
    L = for(1, N, fun() -> spawn(fun() -> wait() end) end),
    {_, Time1} = statistics(runtime),
    {_, Time2} = statistics(wall_clock),
    lists:foreach(fun(Pid) -> Pid ! die end, L),
    U1 = Time1 * 1000 / N,
    U2 = Time2 * 1000 / N,
    io:format("Process spawn time=~p (~p) microseconds~n",
              [U1, U2]).

wait() ->
    receive
        die -> void
    end.

for(N, N, F) -> [F()];
for(I, N, F) -> [F()|for(I+1, N, F)].
```

下面的结果源于我现在所用的计算机（2.90 GHz的Intel Core i7双核处理器，8GB内存，运行Ubuntu操作系统）：

```
1> processes:max(20000).
Maximum allowed processes:262144
Process spawn time=3.0 (3.4) microseconds
2> processes:max(300000).
Maximum allowed processes:262144

=ERROR REPORT==== 14-May-2013::09:32:56 ===
Too many processes

** exception error: a system limit has been reached
...
```

分裂20 000个进程平均消耗了3.0微秒/进程的CPU时间和3.4微秒/进程的实际运行时间。

请注意我使用了内置函数`erlang:system_info(process_limit)`来找出所允许的最大进程数量。其中有一些是系统保留的进程，所以你的程序实际上不能用那么多。当超出限制值时，系统会拒绝启动更多的进程并生成一个错误报告（见第2个命令）。

系统内设的限制值是262 144个进程。要超越这一限制，必须用+P标识启动Erlang仿真器如下：

```
$ erl +P 3000000
1> processes:max(500000).
Maximum allowed processes:4194304
Process spawn time=2.52 (2.896) microseconds
ok
2> processes:max(1000000).
Maximum allowed processes:4194304
Process spawn time=3.65 (4.095) microseconds
ok
3> processes:max(2000000).
Maximum allowed processes:4194304
Process spawn time=4.02 (8.0625) microseconds
ok
6> processes:max(3000000).
Maximum allowed processes:4194304
Process spawn time=4.048 (8.624) microseconds
ok
```

在前面的例子里，系统实际选择的值是恰好大于参数的2的幂。这个实际值可以通过调用`erlang:system_info(process_limit)`获得。我们可以看到，随着进程数量的增加，进程分裂时间也在增加。如果继续增加进程的数量，最终会耗尽物理内存，导致系统开始把物理内存交换到硬盘上，运行速度明显变慢。

如果你编写的程序需要使用大量进程，最好先搞清楚物理内存在交换到硬盘之前能容纳多少进程，并且确保程序运行在物理内存中。

如你所见，创建大量进程的速度是很快的。如果你是一名C或Java程序员，也许会不敢使用大量的进程，而且必须负责管理它们。而在Erlang里，创建进程让编程变得更简单，而不是更复杂。

12.4 带超时的接收

有时候一条接收语句会因为消息迟迟不来而一直等下去。发生这种情况的原因有很多，比如程序里可能有一处逻辑错误，或者准备发送消息的进程在消息发出前就崩溃了。要避免这个问题，可以给接收语句增加一个超时设置，设定进程等待接收消息的最长时间。它的语法如下：

```
receive
    Pattern1 [when Guard1] ->
        Expressions1;
```

```
        Pattern2 [when Guard2] ->
            Expressions2;
        ...
after Time ->
        Expressions
end
```

如果在进入接收表达式的Time毫秒后还没有收到匹配的消息，进程就会停止等待消息，转而执行Expressions。

12.4.1　只带超时的接收

可以编写一个只有超时部分的receive。通过这种方法，我们可以定义一个sleep(T)函数，它会让当前的进程挂起T毫秒。

lib_misc.erl

```
sleep(T) ->
    receive
    after T ->
       true
    end.
```

12.4.2　超时值为 0 的接收

超时值为0会让超时的主体部分立即发生，但在这之前，系统会尝试对邮箱里的消息进行匹配。我们可以用它来定义一个flush_buffer函数，它会清空进程邮箱里的所有消息。

lib_misc.erl

```
flush_buffer() ->
    receive
        _Any ->
            flush_buffer()
    after 0 ->
        true
    end.
```

如果没有超时子句，flush_buffer就会在邮箱为空时永远挂起且不返回。我们还可以使用零超时来实现某种形式的“优先接收”，就像下面这样：

lib_misc.erl

```
priority_receive() ->
    receive
        {alarm, X} ->
            {alarm, X}
    after 0 ->
        receive
```

```
            Any ->
                Any
        end
    end.
```

如果邮箱里不存在匹配{alarm, X}的消息，priority_receive就会接收邮箱里的第一个消息。如果没有任何消息，它就会在最里面的接收语句处挂起，并返回它收到的第一个消息。如果存在匹配{alarm, X}的消息，这个消息就会被立即返回。请记住，只有当邮箱里的所有条目都进行过模式匹配后，才会检查after部分。

如果没有after 0语句，警告（alarm）消息就不会被首先匹配。

注意 对大的邮箱使用优先接收是相当低效的，所以如果你打算使用这一技巧，请确保邮箱不要太满。

12.4.3 超时值为无穷大的接收

如果接收语句里的超时值是原子infinity（无穷大），就永远不会触发超时。这对那些在接收语句之外计算超时值的程序可能很有用。有时候计算的结果是返回一个实际的超时值，其他的时候则是让接收语句永远等待下去。

12.4.4 实现一个定时器

可以用接收超时来实现一个简单的定时器。

函数stimer:start(Time, Fun)会在Time毫秒之后执行Fun（一个不带参数的函数）。它返回一个句柄（是一个PID），可以在需要时用来关闭定时器。

stimer.erl

```
-module(stimer).
-export([start/2, cancel/1]).

start(Time, Fun) -> spawn(fun() -> timer(Time, Fun) end).
cancel(Pid) -> Pid ! cancel.
timer(Time, Fun) ->
    receive
        cancel ->
            void
    after Time ->
            Fun()
    end.
```

可以像下面这样测试它：

```
1> Pid = stimer:start(5000, fun() -> io:format("timer event~n") end).
<0.42.0>
timer event
```

我等待的时间超过了5秒钟，这样定时器就会触发。现在我将启动一个定时器，然后在到期前关闭它。

```
2> Pid1 = stimer:start(25000, fun() -> io:format("timer event~n") end).
<0.49.0>
3> stimer:cancel(Pid1).
cancel
```

超时和定时器是实现许多通信协议的关键。我们等待某个消息时并不想永远等下去，所以会像例子里那样增加一个超时设置。

12.5　选择性接收

基本函数receive用来从进程邮箱里提取消息，但它所做的不仅仅是简单的模式匹配。它还会把未匹配的消息加入队列供以后处理，并管理超时。下面这个语句：

```
receive
    Pattern1 [when Guard1] ->
        Expressions1;
    Pattern2 [when Guard2] ->
        Expressions2;
     ...
after
    Time ->
        ExpressionsTimeout
end
```

它的工作方式如下。

(1) 进入receive语句时会启动一个定时器（但只有当表达式包含after部分时才会如此）。

(2) 取出邮箱里的第一个消息，尝试将它与Pattern1、Pattern2等模式匹配。如果匹配成功，系统就会从邮箱中移除这个消息，并执行模式后面的表达式。

(3) 如果receive语句里的所有模式都不匹配邮箱的第一个消息，系统就会从邮箱中移除这个消息并把它放入一个“保存队列”，然后继续尝试邮箱里的第二个消息。这一过程会不断重复，直到发现匹配的消息或者邮箱里的所有消息都被检查过了为止。

(4) 如果邮箱里的所有消息都不匹配，进程就会被挂起并重新调度，直到新的消息进入邮箱才会继续执行。新消息到达后，保存队列里的消息不会重新匹配，只有新消息才会进行匹配。

(5) 一旦某个消息匹配成功，保存队列里的所有消息就会按照到达进程的顺序重新进入邮箱。如果设置了定时器，就会清除它。

(6) 如果定时器在我们等待消息时到期了，系统就会执行表达式ExpressionsTimeout，并把所有保存的消息按照它们到达进程的顺序重新放回邮箱。

12.6　注册进程

如果想给一个进程发送消息，就需要知道它的PID，但是当进程创建时，只有父进程才知道它的PID。系统里没有其他进程知道它的存在。这通常很不方便，因为你必须把PID发送给系统里所有想要和它通信的进程。另一方面，这也很安全。如果不透露某个进程的PID，其他进程就无法以任何方式与其交互。

Erlang有一种公布进程标识符的方法，它让系统里的任何进程都能与该进程通信。这样的进程被称为*注册进程*（registered process）。管理注册进程的内置函数有四个。

- register(AnAtom, Pid)

  用`AnAtom`（一个原子）作为名称来注册进程`Pid`。如果`AnAtom`已被用于注册某个进程，这次注册就会失败。

- unregister(AnAtom)

  移除与`AnAtom`关联的所有注册信息。

> **注意**　如果某个注册进程崩溃了，就会自动取消注册。

- whereis(AnAtom) -> Pid | undefined

  检查`AnAtom`是否已被注册。如果是就返回进程标识符`Pid`，如果没有找到与`AnAtom`关联的进程就返回原子`undefined`。

- registered() -> [AnAtom::atom()]

  返回一个包含系统里所有注册进程的列表。

可以用`register`来改写12.1节里的代码示例，并尝试用创建的进程名称进行注册。

```
1> Pid = spawn(area_server0, loop, []).
<0.51.0>
2> register(area, Pid).
true
```

一旦名称注册完成，就可以像这样给它发送消息：

```
3> area ! {rectangle, 4, 5}.
Area of rectangle is 20
{rectangle,4,5}
```

可以用`register`来制作一个模拟时钟的注册进程。

clock.erl

```
-module(clock).
-export([start/2, stop/0]).
```

```
start(Time, Fun) ->
    register(clock, spawn(fun() -> tick(Time, Fun) end)).
stop() -> clock ! stop.
tick(Time, Fun) ->
    receive
        stop ->
            void
    after Time ->
            Fun(),
            tick(Time, Fun)
    end.
```

这个时钟会不断滴答作响，直到你停止它。

```
3> clock:start(5000, fun() -> io:format("TICK ~p~n",[erlang:now()]) end).
true
TICK {1164,553538,392266}
TICK {1164,553543,393084}
TICK {1164,553548,394083}
TICK {1164,553553,395064}
4> clock:stop().
stop
```

12.7　关于尾递归的说明

再来看一下我们之前编写的面积计算服务器，它的接收循环如下：

area_server_final.erl

```
loop() ->
    receive
        {From, {rectangle, Width, Ht}} ->
            From ! {self(), Width * Ht},
            loop();
        {From, {circle, R}} ->
            From !  {self(), 3.14159 * R * R},
            loop();
        {From, Other} ->
            From ! {self(), {error,Other}},
            loop()
    end.
```

如果你仔细观察，就会发现每当我们收到消息时就会处理它并立即再次调用`loop()`。这一过程被称为*尾递归*（tail-recursive）。可以对一个尾递归的函数进行特别编译，把语句序列里的最后一次函数调用替换成跳至被调用函数的开头。这就意味着尾递归的函数无需消耗栈空间也能一直循环下去。

假设编写了以下（不正确的）代码：

```
Line 1 loop() ->
-          receive
-              {From, {rectangle, Width, Ht}} ->
-                  From ! {self(), Width * Ht},
5                  loop(),
-                  someOtherFunc();
-              {From, {circle, R}} ->
-                  From !  {self(), 3.14159 * R * R},
-                  loop();
10                 ...
-          end
-      end
```

一个并发程序模板

当我编写并发程序时，几乎总是从下面这样的代码起步：

```
-module(ctemplate).
-compile(export_all).

start() ->
    spawn(?MODULE, loop, []).

rpc(Pid, Request) ->
    Pid ! {self(), Request},
    receive
        {Pid, Response} ->
            Response
    end.

loop(X) ->
    receive
        Any ->
            io:format("Received:~p~n",[Any]),
            loop(X)
    end.
```

接收循环仅仅是一个空循环，它会接收并打印出任何我发给它的消息。在开发程序的过程中，我会开始向一些进程发送消息。因为我一开始没有给接收循环添加能匹配这些消息的模式，所以接收语句底部的代码就会把它们打印出来。每到这个时候，我就会给接收循环添加一个匹配模式并重新运行程序。这一技巧在相当程度上决定了我编写程序的顺序：从一个小程序开始，逐渐扩展它，并在开发过程中不断进行测试。

我们在第5行里调用了`loop()`，但是编译器必然推断出“当我调用`loop()`后必须返回这里，因为我得调用第6行里的`someOtherFunc()`”。于是它把`someOtherFunc`的地址推入栈，然后跳到`loop`的开头。这么做的问题在于`loop()`是永不返回的，它会一直循环下去。所以，每次经过第5行，就会有一个返回地址被推入控制栈，最终系统的空间会消耗殆尽。

避免这个问题的方法很简单，如果你编写的函数F是永不返回的（就像loop()一样），就要确保在调用F之后不再调用其他任何东西，并且别把F用在列表或元组构造器里。

12.8　用 MFA 或 Fun 进行分裂

用显式的模块、函数名和参数列表（称为MFA）来分裂一个函数是确保运行进程能够正确升级为新版模块代码（即使用中被再次编译）的恰当方式。动态代码升级机制不适用于fun的分裂，只能用于带有显式名称的MFA上。更多细节请参阅8.10节的相关内容。

如果你不关心动态代码升级，或者确定程序不会在未来进行修改，就可以使用spawn的spawn(Fun)形式。如果有疑问，就使用spawn(MFA)。

就是这样。现在可以编写并发程序了！

接下来我们将关注错误恢复，了解如何运用三个新的概念（连接、信号和捕捉进程退出）来编写容错的并发程序。这正是下一章的主题。

12.9　练习

(1) 编写一个start(AnAtom, Fun)函数来把spawn(Fun)注册为AnAtom。确保当两个并行进程同时执行start/2时你的程序也能正确工作。在这种情况下，必须保证其中一个进程会成功，而另一个会失败。

(2) 用12.3节里的程序在你的机器上测量一下进程分裂所需的时间。在一张进程数量对进程创建时间的图上进行标绘[1]。你能从中得出什么推论？

(3) 编写一个环形计时测试。创建一个由N个进程组成的环。把一个消息沿着环发送M次，这样总共发送的消息数量是N * M。记录不同的N和M值所花费的时间。

用你熟悉的其他编程语言编写一个类似的程序，然后比较一下结果。写一篇博客，把结果在网上发布出来！

① 标绘是指在图上标注一些点，然后把它们连接起来。——译者注

第 13 章

并发程序中的错误

相对于顺序程序，处理并发程序里的错误涉及一种完全不同的思考方式。在这一章里，我们将根据你在第6章里学到的原则，把这些概念延伸到并发程序上。

我们将介绍错误处理的内在理念，以及关于错误是如何在进程间传播和被其他进程捕捉的细节。最后，用一些小范例作为结尾，它们是编写容错式软件的起点。

设想一个只有单一顺序进程的系统。如果这个进程挂了，麻烦可能就大了，因为没有其他进程能够帮忙。出于这个原因，顺序语言把重点放在故障预防上，强调进行*防御式编程*。

在Erlang里，我们有大量的进程可供支配，因此任何单进程故障都不算特别重要。通常只需编写少量的防御性代码，而把重点放在编写*纠正性代码*上。我们采取各种措施检测错误，然后在错误发生后纠正它们。

13.1 错误处理的理念

并发Erlang程序里的错误处理建立在*远程检测和处理错误*的概念之上。和在发生错误的进程里处理错误不同，我们选择让进程崩溃，然后在其他进程里纠正错误。

在设计容错式系统时就假设错误会发生，进程会崩溃，机器会出故障。我们的任务是在错误发生后检测出来，可能的话还要纠正它们。同时要避免让系统的用户注意到任何的故障，或者在错误修复过程中遭受服务中断。

因为重点在补救而不是预防上，所以系统里几乎没有防御性代码，只有在错误发生后清理系统的代码。这就意味着我们将把注意力放在如何检测错误，如何识别问题来源，以及如何保持系统处于稳定状态上。

检测错误和找出故障原因内建于Erlang虚拟机底层的功能，也是Erlang编程语言的一部分。标准OTP库提供了构建互相监视的进程组和在检测到错误时采取纠正措施的功能，23.5节中会进行相关介绍。这一章介绍的是语言层面的错误检测和恢复。

Erlang关于构建容错式软件的理念可以总结成两个容易记忆的短句：“让其他进程修复错误”和“任其崩溃”。

13.1.1　让其他进程修复错误

我们安排一些进程来互相监控各自的健康状况。如果一个进程挂了，其他某个进程就会注意到并采取纠正措施。

要让一个进程监控另一个，就必须在它们之间创建一个连接（link）或监视（monitor）。如果被连接或监视的进程挂了，监控进程就会得到通知。

监控进程可以实现跨机器的透明运作，因此运行在某一台机器上的进程可以监视运行在不同机器上进程的行为。这是编写容错式系统的基础。不能在一台机器上构建容错式系统，因为崩溃的可能是整台机器，所以至少需要两台机器。一台机器负责计算，其他的机器负责监控它，并在第一台机器崩溃时接管计算。

这可以作为顺序代码错误处理的延伸。虽然可以捕捉顺序代码里的异常并尝试纠正错误（这是第6章的主题），但如果失败了或者整台机器出了故障，就要让其他进程来修复错误。

13.1.2　任其崩溃

如果你来自C这样的语言，这听起来会非常奇怪。在C里，我们被教导要编写防御性代码。程序应当检查它们的参数以避免崩溃。在C里这么做很有必要：编写多进程代码极其困难，而绝大多数应用程序只有一个进程，所以如果这个进程让整个应用程序崩溃，麻烦可就大了。这意味着需要大量的错误检查代码，它们会和非错误检查代码交织在一起。

在Erlang里，我们所做的恰恰相反。我们会把应用程序构建成两个部分：一部分负责解决问题，另一部分负责在错误发生时纠正它们。

负责解决问题的部分会尽可能地少用防御性代码，并假设函数的所有参数都是正确的，程序也会正常运行。

纠正错误的部分往往是通用的，因此同一段错误纠正代码可以用在许多不同的应用程序里。举个例子，如果数据库的某个事务出了错，就简单地中止该事务，让系统把数据库恢复到出错之前的状态。如果操作系统里的某个进程崩溃了，就让操作系统关闭所有打开的文件或套接字，然后让系统恢复到某个稳定状态。

这么做让任务有了清楚的区分。编写解决问题的代码和修复错误的代码，但两者不会交织在一起。代码的体积可能会因此显著变小。

13.1.3　为何要崩溃

让程序在出错时立即崩溃通常是一个很好的主意。事实上，它有不少优点。

- 不必编写防御性代码来防止错误，直接崩溃就好。
- 不必思考应对措施，而是选择直接崩溃，别人会来修复这个错误。
- 不会使错误恶化，因为无需在知道出错后进行额外的计算。
- 如果在错误发生后第一时间举旗示意，就能得到非常好的错误诊断。在错误发生后继续运行经常会导致更多错误发生，让调试变得更加困难。

- 编写错误恢复代码时不用担心崩溃的原因，只需要把注意力放在事后清理上。
- 它简化了系统架构，这样我们就能把应用程序和错误恢复当成两个独立的问题来思考，而不是一个交叉的问题。

相关理念已经介绍得差不多了，现在我们将深入其中的细节。

找其他人来修复它

让别人来修复某个错误而不是自己动手是个不错的主意，能够促进专业化。如果我需要做手术，就会去找大夫，而不是尝试自己操作。

如果我的汽车出了点小问题，车上的控制电脑就会尝试修复它。如果修复失败、问题变得更棘手了，就必须把车拉到修理厂，让其他人来修理它。

如果某个Erlang进程出了点小问题，可以尝试用`catch`或`try`语句来修复它。但如果修复失败，就应该直接崩溃，让其他进程来修复这个错误。

13.2 错误处理的术语含义

在这一节里，你将学到进程间错误处理的术语含义。你会看到一些新名词，在本章的后面还会再次遇到它们。理解错误处理的最佳方式是快速浏览这些名词的定义，然后跳到后面几节，通过更直观的方式理解相关的概念。如果有需要，可以随时查阅本节的内容。

- 进程

  进程有两种：普通进程和系统进程。`spawn`创建的是普通进程。普通进程可以通过执行内置函数`process_flag(trap_exit, true)`变成系统进程。

- 连接

  进程可以互相连接。如果A和B两个进程有连接，而A出于某种原因终止了，就会向B发送一个错误信号，反之亦然。

- 连接组

  进程P的连接组是指与P相连的一组进程。

- 监视

  监视和连接很相似，但它是单向的。如果A监视B，而B出于某种原因终止了，就会向A发送一个“宕机”消息，但反过来就不行了。

- 消息和错误信号

  进程协作的方式是交换消息或错误信号。消息是通过基本函数`send`发送的，错误信号则是进程崩溃或进程终止时自动发送的。错误信号会发送给终止进程的连接组。

- 错误信号的接收

  当系统进程收到错误信号时，该信号会被转换成`{'EXIT', Pid, Why}`形式的消息。`Pid`是终止进程的标识，`Why`是终止原因（有时候被称为退出原因）。如果进程是无错误终止，`Why`就会是原子`normal`，否则`Why`会是错误的描述。

当普通进程收到错误信号时，如果退出原因不是normal，该进程就会终止。当它终止时，同样会向它的连接组广播一个退出信号。

- 显式错误信号

  任何执行exit(Why)的进程都会终止（如果代码不是在catch或try的范围内执行的话），并向它的连接组广播一个带有原因Why的退出信号。

  进程可以通过执行exit(Pid, Why)来发送一个“虚假”的错误信号。在这种情况下，Pid会收到一个带有原因Why的退出信号。调用exit/2的进程则不会终止（这是有意如此的）。

- 不可捕捉的退出信号

  系统进程收到摧毁信号（kill signal）时会终止。摧毁信号是通过调用exit(Pid, kill)生成的。这种信号会绕过常规的错误信号处理机制，不会被转换成消息。摧毁信号只应该用在其他错误处理机制无法终止的顽固进程上。

这些定义可能看上去很复杂，但通常不必深入理解这些机制的工作原理也能编写出容错式代码。系统在错误处理方面的默认行为是尝试“做正确的事”。

后面几节将用一系列的图表来演示错误机制的工作方式。

13.3　创建连接

假设有一组互不相关的进程，如图13-1a所示。虚线代表了连接。

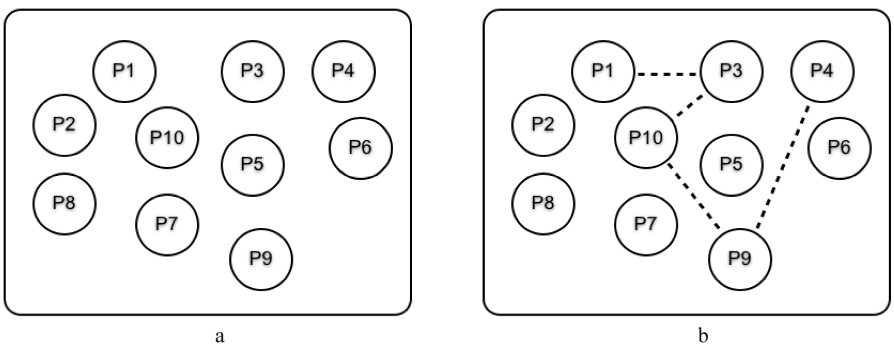


图　13-1

为了创建连接，我们会调用基本函数link(Pid)，它会在调用进程和Pid之间创建一个连接。因此，如果P1调用link(P3)，P1和P3之间就会建立连接。

P1调用了link(P3)，P3又调用了link(P10)，以此类推，最终得到了图13-1b所展示的情形。请注意，P1的连接组只有一个元素（P3），而P3的连接组有两个元素（P1和P10），以此类推。

13.4　同步终止的进程组

通常，你希望创建能够同步终止的进程组。在论证系统行为的时候，这是一个非常有用的不

变法则。当多个进程合作解决问题而某处出现问题时，有时候我们能进行恢复。但如果无法恢复，就会希望之前所做的一切事情都停止下来。它和事务（transaction）这个概念很像：进程要么做它们该做的事，要么全部被杀死。

假设我们有一些相互连接的进程而其中的某个进程挂了，比如图13-2a中的P9。图a展示了P9终止前各个进程是如何连接的。图b展示了P9崩溃且所有错误信号都处理完成后还剩下哪些进程。

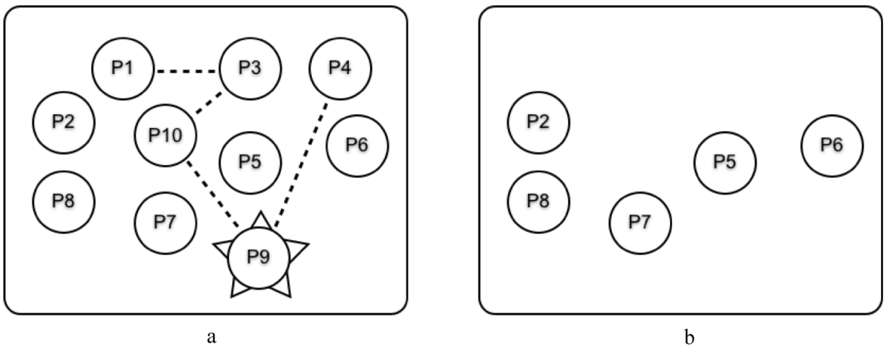


图 13-2

当P9终止时，一个错误信号被发送给进程P4和P10。因为P4和P10不是系统进程，所以也一起终止了，随后，错误信号被发送给与它们相连的所有进程。最后，错误信号扩散到了所有相连的进程，整个互连进程组都终止了。

如果P1、P3、P4、P9或P10里的任意进程终止，它们就会全部终止。

13.5 设立防火墙

有时候我们不希望相连的进程全部终止，而是想让系统里的错误停止扩散。图13-3对此进行了演示，里面所有的相连进程都会终止，一直到P3为止。

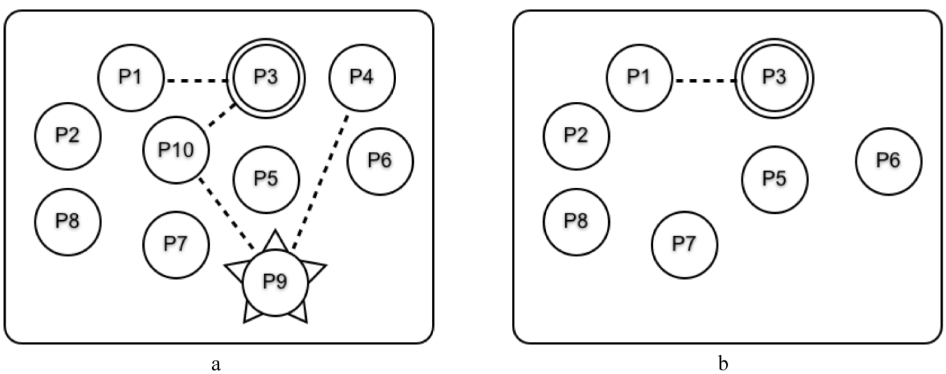


图 13-3

要实现这一点，P3可以执行process_flag(trap_exit, true)并转变成一个系统进程（意思是它可以捕捉退出信号）。如图b所示，它用双圆来表示。P9崩溃之后，错误的扩散会在P3处停止，因此P1和P3不会终止。

P3充当了一个防火墙，阻止错误扩散到系统里的其他进程中。

13.6　监视

监视与连接类似，但是有几处明显的区别。

- 监视是单向的。如果A监视B而B挂了，就会向A发送一个退出消息，反过来则不会如此（别忘了连接是双向的，因此如果A与B相连，其中任何一个进程的终止都会导致另一个进程收到通知）。
- 如果被监视的进程挂了，就会向监视进程发送一个“宕机”消息，而不是退出信号。这就意味着监视进程即使不是系统进程也能够处理错误。

当你想要不对称的错误处理时，可以使用监视，对称的错误处理则适合使用连接。监视通常会被服务器用来监视客户端的行为。

下一节将对操作连接和监视的内置函数进行介绍。

13.7　基本错误处理函数

下列基本函数被用来操作连接和监视，以及捕捉和发送退出信号：

- -spec spawn_link(Fun) -> Pid

  -spec spawn_link(Mod, Fnc, Args) -> Pid

  它们的行为类似于spawn(Fun)和spawn(Mod,Func,Args)，同时还会在父子进程之间创建连接。

- -spec spawn_monitor(Fun) -> {Pid, Ref}

  -spec spawn_monitor(Mod, Func, Args) -> {Pid, Ref}

  它与spawn_link相似，但创建的是监视而非连接。Pid是新创建进程的进程标识符，Ref是该进程的引用。如果这个进程因为Why的原因终止了，消息{'DOWN',Ref,process,Pid,Why}就会被发往父进程。

- -spec process_flag(trap_exit, true)

  它会把当前进程转变成系统进程。系统进程是一种能接收和处理错误信号的进程。

- -spec link(Pid) -> true

  它会创建一个与进程Pid的连接。连接是双向的。如果进程A执行了link(B)，就会与B相连。实际效果就和B执行link(A)一样。

  如果进程Pid不存在，就会抛出一个noproc退出异常。

  如果执行link(B)时A已经连接了B（或者相反），这个调用就会被忽略。

- -spec unlink(Pid) -> true

  它会移除当前进程和进程Pid之间的所有连接。

- -spec erlang:monitor(process, Item) -> Ref

  它会设立一个监视。Item可以是进程的Pid，也可以是它的注册名称。

- -spec demonitor(Ref) -> true

  它会移除以Ref作为引用的监视。

- -spec exit(Why) -> none()

  它会使当前进程因为Why的原因终止。如果执行这一语句的子句不在catch语句的范围内，此进程就会向当前连接的所有进程广播一个带有参数Why的退出信号。它还会向所有监视它的进程广播一个DOWN消息。

- -spec exit(Pid, Why) -> true

  它会向进程Pid发送一个带有原因Why的退出信号。执行这个内置函数的进程本身不会终止。它可以用于伪造退出信号。

  可以用这些基本函数来设立互相监视的进程网络，并把它作为构建容错式软件的起点。

13.8 容错式编程

在这一节里，你将学到一些编写容错式代码的简单技巧。这并不是制作容错式系统的完整教程，而是一个起点。

13.8.1 在进程终止时执行操作

函数on_exit(Pid, Fun)会监视进程Pid，如果它因为原因Why退出了，就会执行Fun(Why)。

lib_misc.erl

```
Line 1 on_exit(Pid, Fun) ->
2          spawn(fun() ->
3                        Ref = monitor(process, Pid),
4                        receive
5                            {'DOWN', Ref, process, Pid, Why} ->
6                                Fun(Why)
7                        end
8                end).
```

monitor(process, Pid)（第3行）对Pid创建了一个监视。当这个进程终止时，监视进程就会接收一个DOWN消息（第5行）并调用Fun(Why)（第6行）。

为了测试它，我们将定义一个F函数，让它等待一个消息X然后计算list_to_atom(X)。

```
1> F = fun() ->
           receive
               X -> list_to_atom(X)
           end
       end.
#Fun<erl_eval.20.69967518>
```

为什么分裂和连接必须是原子操作

以前，Erlang有两个基本函数：spawn和link，而spawn_link(Mod, Func, Args)的定义是下面这样的：

```
spawn_link(Mod, Func, Args) ->
    Pid = spawn(Mod, Fun, Args),
    link(Pid),
    Pid.
```

后来出现了一个诡异的bug。分裂出的进程在连接语句被调用前就挂了，因此进程的退出并没有生成错误信号。这个bug花了好长时间才找到。为了修复这个问题，我们添加了原子操作[①]的spawn_link。只要涉及并发，看似简单的程序也可能会变得很棘手。

我们将分裂出一个进程：

```
2> Pid = spawn(F).
<0.61.0>
```

然后设立一个on_exit处理进程来监视它。

```
3> lib_misc:on_exit(Pid,
                    fun(Why) ->
                        io:format(" ~p died with:~p~n",[Pid, Why])
                    end).
<0.63.0>
```

如果向Pid发送一个原子，这个进程就会挂掉（因为它试图对非列表类型执行list_to_atom），on_exit处理进程则会得到通知。

```
4> Pid ! hello.
hello
5>
=ERROR REPORT==== 14-May-2013::10:05:42 ===
Error in process <0.36.0> with exit value:
   {badarg,[{erlang,list_to_atom,[hello],[]}]}
```

进程挂掉时触发的函数可以执行任何它喜欢的计算：它可以忽略错误，记录错误或者重启应用程序。这个选择完全取决于程序员。

① 原子操作（atomic operation）是指把一个或多个步骤捆绑在一起，成为不可分割的单次操作。——译者注

13.8.2 让一组进程共同终止

假设要创建若干工作进程来解决某个问题，它们分别执行函数F1, F2...。如果任何一个进程挂了，我们希望它们能全体终止。可以调用start([F1,F2, ...])来实现这一点。

```
start(Fs) ->
    spawn(fun() ->
              [spawn_link(F) || F <- Fs],
              receive
                  after
                      infinity -> true
              end
          end).
```

start(Fs)会分裂出一个进程，后者随即分裂并连接各个工作进程，然后无限期等待。如果任何一个工作进程挂了，它们就都会终止。

如果想知道这些进程是否都已终止，可以给启动进程添加一个on_exit处理进程。

```
Pid = start([F1, F2, ...]),
on_exit(Pid, fun(Why) ->
                 ... 如果任何一个工作进程挂了,
                 ... 就运行这里的代码
             end)
```

13.8.3 生成一个永不终止的进程

在这一章的最后，我们将生成一个持久运行的进程。具体做法是生成一个始终活动的注册进程，如果它出于任何原因挂了，就会立即重启。

可以用on_exit来编写它。

lib_misc.erl

```
keep_alive(Name, Fun) ->
    register(Name, Pid = spawn(Fun)),
    on_exit(Pid, fun(_Why) -> keep_alive(Name, Fun) end).
```

这段代码生成一个名为Name的注册进程，并让它执行spawn(Fun)。如果这个进程出于任何原因挂了，就会被重启。

on_exit和keep_alive里有一个相当微妙的错误。如果仔细查看下面这两行代码：

```
Pid = register(...),
on_exit(Pid, fun(X) -> ..),
```

我们就能看到进程有可能会在这两个语句之间挂掉。如果进程在on_exit被执行之前终止，就不会创建连接，on_exit进程的行为就会和预计的不同。如果有两个程序同时尝试用相同的Name值执行keep_alive，这个错误就会发生。这被称为竞争状况（race condition）：两段代码（都

13

是这段）和on_exit里执行连接操作的代码片段正在互相竞争。如果这里出了错，程序就可能会表现出你预料之外的行为。

我不会在此处解决这个问题，而是让你自己思考该怎么做。组合使用spawn、spawn_monitor和register等Erlang基本函数时，必须仔细考虑潜在的竞争状况，确保代码不会让这些状况发生。

现在你已经了解了所有错误处理的知识。顺序代码无法捕捉的错误会从发生错误的进程中流出，沿着连接到达其他进程，而后者可以根据设定来处理这些错误。我们描述过的所有机制（连接过程等）都可以实现跨机器的透明运作。

跨机器运作带领我们进入了分布式编程的领域。Erlang进程可以在网络上的其他物理机器中分裂出新的进程，这让编写分布式程序变得简单了。而这正是下一章的主题。

13.9　练习

(1) 编写一个my_spawn(Mod, Func, Args)函数。它的行为类似spawn(Mod, Func, Args)，但有一点区别。如果分裂出的进程挂了，就应打印一个消息，说明进程挂掉的原因以及在此之前存活了多长时间。

(2) 用本章前面展示的on_exit函数来完成上一个练习。

(3) 编写一个my_spawn(Mod, Func, Args, Time)函数。它的行为类似spawn(Mod, Func, Args)，但有一点区别。如果分裂出的进程存活超过了Time秒，就应当被摧毁。

(4) 编写一个函数，让它创建一个每隔5秒就打印一次“我还在运行”的注册进程。编写一个函数来监视这个进程，如果进程挂了就重启它。启动公共进程和监视进程，然后摧毁公共进程，检查它是否会被监视进程重启。

(5) 编写一个函数来启动和监视多个工作进程。如果任何一个工作进程非正常终止，就重启它。

(6) 编写一个函数来启动和监视多个工作进程。如果任何一个工作进程非正常终止，就摧毁所有工作进程，然后重启它们。

第 14 章

分布式编程

用Erlang编写分布式程序和编写并发程序只有一步之遥。在分布式Erlang里，可以在远程节点和机器上分裂进程。分裂出远程进程之后，我们会看到其他所有的基本函数（`send`、`receive`和`link`等）都能透明运作在网络中，就像在单个节点上一样。

在本章，我们将介绍用于编写分布式Erlang程序的库与Erlang基本函数。分布式程序是那些被设计运行在计算机网络上的程序，并且可以仅靠传递消息来协调彼此的活动。

下面是一些想要编写分布式应用程序的原因。

- 性能

  可以通过安排程序的不同部分在不同的机器上并行运行来让程序跑得更快。
- 可靠性

  可以通过让系统运行在数台机器上来实现容错式系统。如果一台机器出了故障，可以在另一台机器上继续。
- 可扩展性

  随着我们把应用程序越做越大，即使机器的处理能力再强大也迟早会耗尽。到那时，就必须添加更多的机器来提升处理能力。添加一台新机器应当是一次简单的操作，不需要对应用程序的架构做出大的修改。
- 天生分布式的程序

  许多应用程序天生就是分布式的。如果编写一个多用户游戏或聊天系统，就会有来自世界各地的分散用户。如果我们在某个地理位置上拥有大量的用户，就会希望把计算资源放置在接近这些用户的地方。
- fun

  我想要编写的fun程序大部分都是分布式的。其中许多涉及与全世界各地的人与机器进行交互。

14.1 两种分布式模型

我们在本书里将讨论两种主要的分布式模型。

- 分布式Erlang

  在分布式Erlang里，我们编写的程序会在Erlang的节点（node）上运行。节点是一个独立

的Erlang系统，包含一个自带地址空间和进程组的完整虚拟机。

可以在任何节点上分裂进程，前几章讨论的所有消息传递和错误处理基本函数也都能像在单节点上那样工作。

分布式Erlang应用程序运行在一个可信环境中。因为任何节点都可以在其他Erlang节点上执行任意操作，所以这涉及高度的信任。虽然分布式Erlang应用程序可以运行在开放式网络上，但它们通常是运行在属于同一个局域网的集群上，并受防火墙保护。

- 基于套接字的分布式模型

  可以用TCP/IP套接字来编写运行在不可信环境中的分布式应用程序。这个编程模型不如分布式Erlang那样强大，但是更安全。在14.6节里，我们会来看看如何用基于套接字的简单分布式机制来构建应用程序。

如果你回想一下前几章的内容，就一定还记得我们构建程序的基本单位是进程。编写分布式Erlang程序是很容易的，要做的就是在正确的机器上分裂出进程，然后一切就能像之前那样运作了。

我们都习惯了编写顺序程序，而编写分布式程序通常会困难得多。在这一章里，我们将介绍编写简单分布式程序的若干技巧。这些程序很简单，但是非常有用。

我们将从一些小范例起步。只需先学习两件事，就可以开始创建我们的第一个分布式程序了。我们将学习如何启动一个Erlang节点，以及如何在远程Erlang节点上执行远程过程调用。

14.2　编写一个分布式程序

当我开发一个分布式应用程序时，总是会按照特定的顺序来编写它。

(1) 在一个常规的非分布式会话里编写和测试我的程序。这是我们到目前为止一直在做的，所以不会有什么新问题。

(2) 在运行于同一台计算机上的两个不同的Erlang节点里测试程序。

(3) 在运行于两台物理隔离计算机上的两个不同的Erlang节点里测试程序，这两台计算机或者属于同一个局域网，或者来自互联网的任何地方。

最后一步可能会带来问题。如果所运行的机器属于相同的管理域，就很少会出问题。但当相关节点属于不同域上的机器时，我们就可能会遇到连接性问题，而且必须确保系统防火墙和安全设置都已得到正确配置。

为了演示这些步骤，我们将制作一个简单的名称服务器（name server）。具体而言，将执行下列步骤。

- 第1阶段：在一个常规的非分布式Erlang系统上编写和测试名称服务器。
- 第2阶段：在同一台机器的两个节点上测试名称服务器。
- 第3阶段：在同一局域网内分属两台不同机器的节点上测试名称服务器。
- 第4阶段：在分属两个不同国家和域的两台机器上测试名称服务器。

14.3　创建名称服务器

名称服务器这种程序会返回一个给定名称的关联值。我们也可以修改某个名称所关联的值。

我们的第一个名称服务器极其简单。它不是容错式的，所以如果它崩溃了，保存的数据就会全部丢失。这个练习的目的不是创建一个容错式名称服务器，而是开始运用分布式编程的技巧。

14.3.1　第 1 阶段：一个简单的名称服务器

我们的名称服务器kvs是一个简单的Key → Value服务器，它的接口如下。

- *-spec kvs:start() -> true*

  启动服务器。它将创建一个注册名为kvs的服务器。

- *-spec kvs:store(Key, Value) -> true*

  关联Key和Value。

- *-spec kvs:lookup(Key) -> {ok, Value} | undefined*

  查询Key的值。如果Key带有关联值就返回{ok, Value}，否则返回undefined。

这个键–值服务器是用进程字典里的基本函数get和put实现的，它的代码如下：

socket_dist/kvs.erl

```
-module(kvs).
-export([start/0, store/2, lookup/1]).

start() -> register(kvs, spawn(fun() -> loop() end)).

store(Key, Value) -> rpc({store, Key, Value}).

lookup(Key) -> rpc({lookup, Key}).

rpc(Q) ->
    kvs ! {self(), Q},
    receive
        {kvs, Reply} ->
            Reply
    end.

loop() ->
    receive
        {From, {store, Key, Value}} ->
            put(Key, {ok, Value}),
            From ! {kvs, true},
            loop();
```

```
-           {From, {lookup, Key}} ->
-               From ! {kvs, get(Key)},
25              loop()
-       end.
```

保存（store）消息在第6行发送并在第19行接收。主服务器在第17行的loop函数中启动。它调用了receive并等待一个保存或查询消息，然后保存数据或从本地进程字典里取出被请求的数据，并向客户端发送回复。首先将在本地测试这个服务器，看看它是否能正常工作。

```
1> kvs:start().
true
2> kvs:store({location, joe}, "Stockholm").
true
3> kvs:store(weather, raining).
true
4> kvs:lookup(weather).
{ok,raining}
5> kvs:lookup({location, joe}).
{ok,"Stockholm"}
6> kvs:lookup({location, jane}).
undefined
```

到目前为止，没有什么意料之外的事情发生。下面进入第2个步骤，把这个应用程序分布化。

14.3.2　第 2 阶段：客户端在一个节点，服务器在相同主机的另一个节点

现在在同一台计算机上启动两个Erlang节点。为此，需要打开两个终端窗口，然后启动两套Erlang系统。

首先，将开启一个终端shell，并在这个shell里启动一个名为gandalf的分布式Erlang节点。然后启动服务器：

```
$ erl -sname gandalf
(gandalf@localhost) 1>  kvs:start().
true
```

参数-sname gandalf的意思是“在本地主机上启动一个名为gandalf的Erlang节点”。注意一下Erlang shell是如何把Erlang节点名打印在命令提示符前面的。节点名的形式是Name@Host。Name和Host都是原子，所以如果它们包含任何非原子的字符，就必须加上引号。

重要提示　如果你在自己的系统上运行上面的命令，节点名可能就不是gandolf@localhost，而是gandolf@H（如果H是你的本地主机名的话）。它会根据你的系统配置而定。如果情况确实如此，就必须在接下来所有的范例里用名称H代替localhost。

接下来将开启第二个终端会话，然后启动一个名为bilbo的Erlang节点。这样就可以用库模块rpc来调用kvs里的函数了。（请注意，rpc是一个标准的Erlang库模块，和之前编写的rpc函数不是一回事。）

```
$ erl -sname bilbo
(bilbo@localhost) 1> rpc:call(gandalf@localhost,
    kvs,store, [weather, fine]).
true
(bilbo@localhost) 2> rpc:call(gandalf@localhost,
    kvs,lookup,[weather]).
{ok,fine}
```

虽然看起来不太起眼，但实际上已经执行了我们的第一次分布式计算！服务器运行在我们启动的第一个节点上，客户端则运行在第二个节点上。

设置weather值的调用是由bilbo节点发出的，可以切换回gandalf来检查一下天气（weather）的值。

```
(gandalf@localhost) 2> kvs:lookup(weather).
{ok,fine}
```

rpc:call(Node, Mod, Func, [Arg1, Arg2, ..., ArgN])会在Node上执行一次远程过程调用。调用的函数是Mod:Func(Arg1, Arg2, ..., ArgN)。

如你所见，这个程序的工作方式和非分布式Erlang一致。唯一的区别在于客户端运行在一个节点上，而服务器运行在另一个不同的节点上。

下一步是在不同的机器上运行客户端和服务器。

14.3.3 第 3 阶段：同一局域网内不同机器上的客户端和服务器

我们将使用两个节点。第一个名为gandalf的节点在doris.myerl.example.com上，第二个名为bilbo的节点在george.myerl.example.com上。开始工作之前，我们先用ssh或vnc等工具在两台不同的机器上各启动一个终端。我们把这两个窗口称为doris和george。做完这些之后，我们就可以在两台机器上轻松输入命令了。

第1步是在doris上启动一个Erlang节点。

```
doris $ erl -name gandalf -setcookie abc
(gandalf@doris.myerl.example.com) 1> kvs:start().
true
```

第2步是在george上启动一个Erlang节点并向gandalf发送一些命令。

```
george $ erl -name bilbo -setcookie abc
(bilbo@george.myerl.example.com) 1> rpc:call(gandalf@doris.myerl.example.com,
    kvs,store,[weather,cold]).
true
(bilbo@george.myerl.example.com) 2> rpc:call(gandalf@doris.myerl.example.com,
    kvs,lookup,[weather]).
{ok,cold}
```

它们的行为和同一机器上两个不同节点的情况完全一致。

要实现这一切，我们的操作会比在同一台机器上运行两个节点时略微复杂一些。我们必须分4步走。

(1) 用`-name`参数启动Erlang。我们在同一台机器上运行两个节点时使用了“短”（short）名称（通过`-sname`标识体现）。但如果它们属于不同的网络，我们就要使用`-name`。

当两台机器位于同一个子网时我们也可以使用`-sname`。而且如果没有DNS服务，`-sname`就是唯一可行的方式。

(2) 确保两个节点拥有相同的`cookie`。这正是启动两个节点时都使用命令行参数`-setcookie abc`的原因。我们会在14.5节对它做更详细的介绍。

注意　当我们在同一台机器上运行两个节点时，因为它们都能访问同一个cookie文件`$HOME/.erlang.cookie`，所以我们不需要在Erlang命令行里添加cookie。

---

(3) 确保相关节点的完全限定主机名（fully qualified hostname）可以被DNS解析。对于我来说，域名`myerl.example.com`完全属于我的家庭网络，通过在`/etc/hosts`里添加一个条目来实现本地解析。

(4) 确保两个系统拥有相同版本的代码和相同版本的Erlang。如果不这么做，就可能会得到严重而离奇的错误。避免问题的最简单的方法是在所有地方都运行相同版本的Erlang。不同版本的Erlang可以一起运行，但是无法保证能正常工作，所以最好事先检查一下。在我们这个案例里，相同版本的`kvs`代码必须存在于两个系统中。通过下列方法可以做到这一点。

- 我家里有两台物理隔离的计算机，它们没有共享的文件系统。在那里我先手工把`kvs.erl`复制到这两台计算机上，编译它们之后再启动程序。
- 我的工作计算机是带有一个共享NFS磁盘的工作站。在那里我只需要让两台工作站从共享目录里启动Erlang就行了。
- 配置代码服务器来实现这一点。我不会在这里介绍具体做法，你可以去看看`erl_prim_loader`模块的手册页。
- 使用shell命令`command nl(Mod)`。它会在所有的相连节点上载入`Mod`模块。

注意　要使用这个命令，必须确保所有节点都已连接。节点会在它们第一次尝试互相访问时建立连接。当首次执行与某个远程节点相关的任意表达式时，它就会发生。最简单的做法是执行`net_adm:ping(Node)`（详情请参见`net_adm`的手册页）。

成功了！我们正在同一局域网的两台服务器上运行。下一步是把它们移到两台通过互联网相连的计算机上。

14.3.4　第 4 阶段：跨互联网不同主机上的客户端和服务器

原则上，这和第3阶段是一样的，但现在我们必须更加关注安全性。运行同一局域网内的两个节点时，多半不会过于担心安全性。在大多数机构里，局域网都是通过防火墙与互联网隔离的。可以在防火墙后面自由分配临时IP地址，对机器的设置也很随意。

当我们跨互联网连接Erlang集群里的几台机器时，可以预料到会出现防火墙不允许传入连接的问题。必须正确配置防火墙，让它接受传入连接。这一点没有通用的做法，因为每一种防火墙都是不同的。

要让系统准备好运行分布式Erlang，需执行以下步骤。

(1) 确保4369端口对TCP和UDP流量都开放。这个端口会被一个名为epmd的程序使用（它是Erlang Port Mapper Daemon的缩写，即Erlang端口映射守护进程）。

(2) 选择一个或一段连续端口给分布式Erlang使用，并确保这些端口是开放的。如果这些端口位于Min和Max之间（只想用一个端口就让Min=Max），就用以下命令启动Erlang：

```
$ erl -name ... -setcookie ... -kernel inet_dist_listen_min Min \
                                       inet_dist_listen_max Max
```

现在，我们已经了解了如何在一组Erlang节点上运行程序，以及如何通过局域网和互联网运行它们。下面来看看操作节点的基本函数。

14.4 分布式编程的库和内置函数

我们在编写分布式程序时很少从头开始。标准库里有许多模块可以用于编写分布式程序。虽然这些模块是用内置分布式函数编写的，但是它们能对程序员隐藏大量繁琐的细节。

标准分发套装里的两个模块能够满足大多数需求。

- ❑ rpc提供了许多远程过程调用服务。
- ❑ global里的函数可以用来在分布式系统里注册名称和加锁，以及维护一个全连接网络。

rpc模块里最重要的函数就是下面这个。

call(Node, Mod, Function, Args) -> Result | {badrpc, Reason}

它会在Node上执行apply(Mod, Function, Args)，然后返回结果Result，如果调用失败则返回{badrpc, Reason}。

以下是编写分布式程序的基本函数（关于这些内置函数的更完整的介绍请参见erlang模块的手册页[1]）。

- -spec spawn(Node, Fun) -> Pid

  它的工作方式和spawn(Fun)完全一致，只是新进程是在Node上分裂的。

- -spec spawn(Node, Mod, Func, ArgList) -> Pid

  它的工作方式和spawn(Mod, Func, ArgList)完全一致，只是新进程是在Node上分裂的。spawn(Mod, Func, Args)会创建一个执行apply(Mod, Func, Args)的新进程。它会返回这个新进程的PID。

① http://www.erlang.org/doc/man/erlang.html

> **注意**　这种形式的spawn比spawn(Node, Fun)更加健壮。如果运行在多个分布式节点上的特定模块不是完全相同的版本，spawn(Node, Fun)就可能会出错。

- -spec spawn_link(Node, Fun) -> Pid

  它的工作方式和spawn_link(Fun)完全一致，只是新进程是在Node上分裂的。

- -spec spawn_link(Node, Mod, Func, ArgList) -> Pid

  它的工作方式类似spawn(Node, Mod, Func, ArgList)，但是新进程会与当前进程相连。

- -spec disconnect_node(Node) -> bool() | ignored

  它会强制断开与某个节点的连接。

- -spec monitor_node(Node, Flag) -> true

  如果Flag是true就会开启监视，Flag是false就会关闭监视。如果开启了监视，那么当Node加入或离开Erlang互连节点组时，执行这个内置函数的进程就会收到{nodeup, Node}或{nodedown, Node}的消息。

- -spec node() -> Node

  它会返回本地节点的名称。如果节点不是分布式的则会返回nonode@nohost。

- -spec node(Arg) -> Node

  它会返回Arg所在的节点。Arg可以是PID、引用或者端口。如果本地节点不是分布式的，则会返回nonode@nohost。

- -spec nodes() -> [Node]

  它会返回一个列表，内含网络里其他所有与我们相连的节点。

- -spec is_alive() -> bool()

  如果本地节点是活动的，并且可以成为分布式系统的一部分，就返回true，否则返回false。

  另外，send可以用来向一组分布式Erlang节点里的某个本地注册进程发送消息。下面的语法：

  ```
  {RegName, Node} ! Msg
  ```

  可以把消息Msg发送给节点Node上的注册进程RegName。

14.4.1　远程分裂示例

作为一个简单的示例，我们将展示如何在某个远程节点上分裂进程。先从下面这个程序开始。

dist_demo.erl

```
-module(dist_demo).

-export([rpc/4, start/1]).

start(Node) ->
    spawn(Node, fun() -> loop() end).

rpc(Pid, M, F, A) ->
    Pid ! {rpc, self(), M, F, A},
    receive
        {Pid, Response} ->
            Response
    end.
loop() ->
    receive
        {rpc, Pid, M, F, A} ->
            Pid ! {self(), (catch apply(M, F, A))},
            loop()
    end.
```

然后启动两个节点，它们都必须能够载入这段代码。如果这两个节点在同一台主机上，这就不成问题。只需从同一个目录里启动两个Erlang节点就可以了。

如果节点分别属于两台物理隔离且文件系统不同的主机，这个程序就必须被复制到所有节点上，编译之后才能启动节点（或者也可以把.beam文件复制到所有节点上）。在这个例子里，我假定这一切都已完成。

在主机doris上，启动一个名为gandalf的节点。

```
doris $ erl -name gandalf -setcookie abc
(gandalf@doris.myerl.example.com) 1>
```

在主机george上，启动一个名为bilbo的节点，要记得使用同一个cookie。

```
george $ erl -name bilbo -setcookie abc
(bilbo@george.myerl.example.com) 1>
```

现在（在bilbo上），让远程节点（gandalf）分裂一个进程。

```
(bilbo@george.myerl.example.com) 1> Pid =
          dist_demo:start('gandalf@doris.myerl.example.com').
<5094.40.0>
```

现在，Pid是这个远程节点进程的标识符，调用dist_demo:rpc/4，在远程节点上执行一次远程过程调用。

```
(bilbo@george.myerl.example.com) 2> dist_demo:rpc(Pid, erlang, node, []).
'gandalf@doris.myerl.example.com'
```

它在远程节点上执行erlang:node()并返回一个值。

14.4.2　文件服务器再探

我们在2.3.1节创建了一个文件服务器，并且承诺稍后会再次讨论它，也就是指现在了。本章前面部分展示了如何设立一个简单的远程过程调用服务器，它可用来在两个Erlang节点之间传输文件。

下面这些操作是上一个示例的延续。

```
(bilbo@george.myerl.example.com) 1> Pid =
    dist_demo:start('gandalf@doris.myerl.example.com').
<6790.42.0>
(bilbo@george.myerl.example.com) 2>
    dist_demo:rpc(Pid, file, get_cwd, []).
{ok,"/home/joe/projects/book/jaerlang2/Book/code"}
(bilbo@george.myerl.example.com) 3>
    dist_demo:rpc(Pid, file, list_dir, ["."]).
{ok,["adapter_db1.erl","processes.erl",
     "counter.beam","attrs.erl","lib_find.erl",...]}
(bilbo@george.myerl.example.com) 4>
    dist_demo:rpc(Pid, file, read_file, ["dist_demo.erl"]).
{ok,<<"-module(dist_demo).\n-export([rpc/4, start/1]).\n\n...>>}
```

我在`gandalf`上启动了一个分布式Erlang节点，位置是我保存本书代码示例的`code`目录。我在`bilbo`上发起的一些请求形成了对`gandalf`上标准库的远程过程调用。我使用`file`模块里的三个函数来访问`gandalf`的文件系统。`file:get_cwd()`返回文件服务器的当前工作目录，`file:list_dir(Dir)`返回`Dir`里所有文件的列表，`file:read_file(File)`读取文件`File`。

仔细回味一下，你会意识到刚才所做的相当神奇。我们没有编写任何代码就创建了一个文件服务器。只是重用了`file`模块里的库代码，并使它可以通过一个简单的远程过程调用接口访问。

实现一个文件传输程序

几年前，我需要在操作系统不同的两台联网机器之间传输许多文件。我的第一个想法是使用FTP，但这样我就需要在一台机器上设置FTP服务器，在另一台上设置FTP客户端。我找不到能用于服务端机器的FTP服务器软件，而且也没有安装FTP服务器所需的root权限。我有的是在两台机器上运行的分布式Erlang。

我随后使用的正是这里描述的技巧。不做不知道，编写我自己的文件服务器甚至比搜索并安装一个FTP服务器更快。

如果有兴趣，可以看看当时我写的博客[①]。

① http://armstrongonsoftware.blogspot.com/2006/09/why-i-often-implement-things-from.html

14.5 cookie 保护系统

cookie系统让访问单个或一组节点变得更安全。每个节点都有一个cookie，如果它想与其他任何节点通信，它的cookie就必须和对方节点的cookie相同。为了确保cookie相同，分布式Erlang系统里的所有节点都必须以相同的“神奇”（magic）cookie启动，或者通过执行`erlang:set_cookie`把它们的cookie修改成相同的值。

Erlang集群的定义就是一组带有相同cookie的互连节点。

两个分布式Erlang节点要相互通信，就必须拥有相同的神奇cookie。可以用三种方法设置cookie。

注意 cookie保护系统被设计用来创建运行在局域网（LAN）上的分布式系统，LAN本身应该受防火墙保护，与互联网隔开。跨互联网运行的分布式Erlang应用程序应该先在主机之间建立安全连接，然后再使用cookie保护系统。

- 方法1：在文件`$HOME/.erlang.cookie`里存放相同的cookie。这个文件包含一个随机字符串，是Erlang第一次在你的机器上运行时自动创建的。

  这个文件可以被复制到所有想要参与分布式Erlang会话的机器上。也可以显式设置它的值。举个例子，我们可以在Linux系统上输入下列命令：

  ```
  $ cd
  $ cat > .erlang.cookie
  AFRTY12ESS3412735ASDF12378
  $ chmod 400 .erlang.cookie
  ```

  `chmod`让`.erlang.cookie`文件只能被它的所有者访问。
- 方法2：当Erlang启动时，可以用命令行参数`-setcookie C`来把神奇cookie设成`C`。这里有一个例子：

  ```
  $ erl -setcookie AFRTY12ESS3412735ASDF12378 ...
  ```
- 方法3：内置函数`erlang:set_cookie(node(), C)`能把本地节点的cookie设成原子`C`。

注意 如果你的环境不够安全，那么方法1和3要优于方法2，因为Unix系统里的任何用户都可以用`ps`命令来查看你的cookie。方法2只适用于测试。

另外，cookie从不会在网络中明文传输，它只用来对某次会话进行初始认证。分布式Erlang会话不是加密的，但可以被设置成在加密通道中运行。（可以在Erlang邮件列表里搜索这方面的最新信息。）

到目前为止，我们已经看到了如何用Erlang节点和基本的分布式函数来编写分布式程序。作为替代方案，也可以编写基于原始套接字接口的分布式程序。

14.6　基于套接字的分布式模型

在这一节里，我们将编写一个基于套接字的简单分布式程序。如你所见，分布式Erlang适合编写那些可信任其他参与者的集群应用程序，但在并非人人都可信的开放式环境里，它就不那么适合了。

分布式Erlang的主要问题在于客户端可以自行决定在服务器上分裂出各种进程。因此，要摧毁你的系统，只需执行下面的命令：

```
rpc:multicall(nodes(), os, cmd, ["cd /; rm -rf *"])
```

分布式Erlang的适用情形是你拥有全部的机器，并且想在单台机器上控制它们。但如果这些机器分别为不同的人所有，并且他们想要精确控制自己的机器可以运行哪些软件，这种计算模型就不适合了。

在这种情况下，我们将使用一种受限形式的`spawn`，让机器的所有者能够显式控制自己机器上运行的程序。

14.6.1　用lib_chan控制进程

`lib_chan`模块让用户能够显式控制自己的机器能分裂出哪些进程。`lib_chan`的实现相当复杂，所以我把它从普通章节里单独摘出来。你可以在附录B里找到它。它的接口如下。

- -spec start_server() -> true

  它会在本地主机上启动一个服务器。这个服务器的行为由文件`$HOME/.erlang_config/lib_chan.conf`决定。

- -spec start_server(Conf) -> true

  它会在本地主机上启动一个服务器。这个服务器的行为由文件`Conf`决定，它包含一个由下列形式的元组所组成的列表：

  - {port, NNNN}

    它会开始监听端口号NNNN。
  - {service, S, password, P, mfa, SomeMod, SomeFunc, SomeArgsS}

    它会定义一个被密码P保护的服务S。如果这个服务启动了，就会通过分裂`SomeMod:SomeFunc(MM, ArgsC, SomeArgsS)`创建一个进程，负责处理来自客户端的消息。这里的MM是一个代理进程的PID，可以用来向客户端发送消息。参数`ArgsC`来自于客户端的连接调用。

- -spec connect(Host, Port, S, P, ArgsC) -> {ok, Pid} | {error, Why}

  尝试开启主机`Host`上的端口`Port`，然后尝试激活被密码P保护的服务`S`。如果密码正确，就会返回`{ok, Pid}`。`Pid`是一个代理进程的标识符，可以用来向服务器发送消息。

当客户端调用connect/5建立连接后，就会分裂出两个代理进程，一个在客户端，另一个在服务器端。这些代理进程负责把Erlang消息转换成TCP包数据，捕捉来自控制进程的退出信号，以及套接字关闭。

此番解释可能看上去很复杂，但使用之后就会变得明朗许多。下面这个完整的例子展示了如何将lib_chan与之前描述的kvs结合使用。

14.6.2　服务器代码

首先来编写一个配置文件。

```
{port, 1234}.
{service, nameServer, password, "ABXy45",
         mfa, mod_name_server, start_me_up, notUsed}.
```

它的意思是我们将在自己机器的1234端口上提供一个名为nameServer的服务。这个服务被密码ABXy45保护。

当客户端调用下面的函数来创建连接时：

```
connect(Host, 1234, nameServer, "ABXy45", nil)
```

服务器会分裂mod_name_server:start_me_up(MM, nil, notUsed)。MM是一个代理进程的PID，用来和客户端通信。

注意　*在这个阶段，你应当仔细研读上一行代码，确保明白调用里的这些参数来自哪里。*

- mod_name_server、start_me_up和notUsed来自于配置文件。
- nil是connect调用的最后一个参数。

mod_name_server的代码如下：

socket_dist/mod_name_server.erl

```
-module(mod_name_server).
-export([start_me_up/3]).

start_me_up(MM, _ArgsC, _ArgS) ->
    loop(MM).

loop(MM) ->
    receive
        {chan, MM, {store, K, V}} ->
            kvs:store(K, V),
            loop(MM);
        {chan, MM, {lookup, K}} ->
            MM ! {send, kvs:lookup(K)},
            loop(MM);
```

```
        {chan_closed, MM} ->
            true
    end.
```

mod_name_server遵循以下协议。

- 如果客户端向服务器发送一个消息{send, X}，这个消息在mod_name_server里就会变成{chan, MM, X}的形式（MM是服务器代理进程的PID）。
- 如果客户端终止或者用于通信的套接字出于任何原因关闭了，服务器就会收到一个{chan_closed, MM}形式的消息。
- 如果服务器想给客户端发送一个消息X，就会通过调用MM ! {send, X}实现。
- 如果服务器想要显式关闭连接，就会通过执行MM ! close实现。

这个协议是一个中间人协议，客户端代码和服务器代码都遵循它。本书附录B里的“lib_chan_mm：中间人”一节会更详细地解释套接字中间人代码。

为了测试这段代码，首先会确保它能在单台机器上正常工作。

现在可以启动名称服务器（和kvs模块）了。

```
1> kvs:start().
true
2> lib_chan:start_server().
Starting a port server on 1234...
true
```

现在启动第二个Erlang会话，用任意客户端来测试它。

```
1> {ok, Pid} = lib_chan:connect("localhost",1234,nameServer,"ABXy45","").
{ok, <0.43.0>}
2> lib_chan:cast(Pid, {store, joe, "writing a book"}).
{send,{store,joe,"writing a book"}}
3> lib_chan:rpc(Pid, {lookup, joe}).
{ok,"writing a book"}
4> lib_chan:rpc(Pid, {lookup, jim}).
undefined
```

在单台机器上测试成功之后，我们会按照之前描述的步骤，在两台物理隔离的机器上执行类似测试。

请注意，在这个案例里，决定配置文件内容的是远程机器的所有者。配置文件指定了哪些应用程序是这台机器允许运行的，以及哪个端口是用来与这些应用程序通信的。

现在已经能够编写分布式程序了。一个崭新的世界出现在我们面前。如果编写顺序程序是一种乐趣，那么编写分布式程序就是乐趣的平方或者立方。强烈建议你完成下面的YAFS练习，它的基本代码结构是许多应用程序的中心。

我们已经介绍了顺序、并发和分布式编程。在本书下一部分，我们将来看看如何建立外部语言代码的接口，并介绍一些主要的Erlang库，以及如何调试代码，然后了解如何用OTP构建原则和库来创建复杂的Erlang系统。

14.7　练习

(1) 在同一主机上启动两个节点。查询`rpc`模块的手册页。对这两个节点执行一些远程过程调用。

(2) 重复上一个练习，这次使用同一局域网里的两个节点。

(3) 重复上一个练习，这次使用不同网络里的两个节点。

(4) 用`lib_chan`里的库编写YAFS（Yet Another File Server的缩写，即“又一个文件服务器”）。你会从中学到很多知识。给你的文件服务器添加一些“装饰品”。

Part 4 第四部分

编程库与框架

本书的这一部分将涉及文件、套接字和数据库编程所用到的主要 Erlang 库。此外，我们还会介绍调试技巧和 OTP 框架。

第 15 章

接口技术

系统的构建经常涉及建立接口，用它将不同语言编写的应用程序与我们的系统相连。我们可能会使用C来提高效率或编写底层硬件驱动，或者集成一个Java、Ruby或其他语言编写的库。可以用多种方式建立外部语言程序与Erlang之间的接口。

- 让程序以外部操作系统进程的形式在Erlang虚拟机*以外*运行。这是一种*安全*的做法。即使外部语言的代码有问题，也不会让Erlang系统崩溃。Erlang通过一种名为*端口*（port）的对象来控制外部进程，与外部进程的通信则是通过一个面向字节的通信信道。Erlang负责启动和停止外部程序，还可以监视它，让在它崩溃后重启。外部进程被称为*端口进程*，因为它是通过一个Erlang端口控制的。
- 在Erlang内部运行操作系统命令并捕捉结果。
- 在Erlang虚拟机的*内部*运行外部语言代码。这涉及链接外部代码和Erlang虚拟机代码，是一种*不安全*的做法。外部语言代码里的错误可能会导致Erlang系统崩溃。虽然它不安全，但还是有用的，因为这么做比使用外部进程更高效。

  把代码链接到Erlang内核只适用于C这样能生成本地目标代码的语言，不适用于Java这样自身拥有虚拟机的语言。

在这一章里，我们将介绍如何用端口和操作系统命令建立Erlang接口。另外还有一些使用内链驱动（linked-in driver）、原生实现函数（natively implemented function，简称NIF）和C-node的高级接口技术。本书不会涉及这些高级接口技术，但会在本章后面简要概述它们，并提供一些参考资料指引。

15.1 Erlang 如何与外部程序通信

Erlang通过名为*端口*的对象与外部程序通信。如果向端口发送一个消息，此消息就会被发往与端口相连的外部程序。来自外部程序的消息则会变成来自端口的Erlang消息。

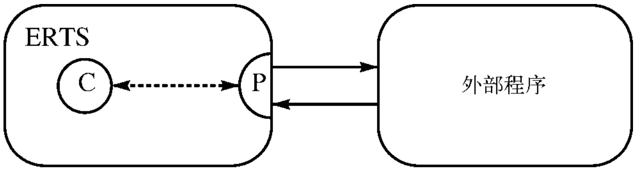
对程序员而言，端口的行为就像是一个Erlang进程。你可以向它发送消息，可以注册它（就像进程一样），诸如此类。如果外部程序崩溃了，就会有一个退出信号发送给相连的进程。如果相连的进程挂了，外部程序就会被关闭。

请注意使用端口与外部进程通信和使用套接字的区别。如果使用端口，它会表现得像一个

Erlang进程，这样就可以链接它，从某个远程分布式Erlang节点向它发送消息，等等。如果使用套接字，就不会表现出类似进程的行为。

创建端口的进程被称为该端口的相连进程。相连进程有其特殊的重要性：所有发往端口的消息都必须标明相连进程的PID，所有来自外部程序的消息都会发往相连进程。

在下面这张图里，我们可以看到相连进程（C）、端口（P）和外部操作系统进程之间的关系。



ERTS = Erlang运行时系统

C = 与端口相连的Erlang进程

P = 端口

要创建一个端口，就会调用`open_port`，它的说明如下。

```
-spec open_port(PortName, [Opt]) -> Port
```

其中`PortName`是下列选项中的一个。

- `{spawn, Command}`
  启动一个外部程序。`Command`是这个外部程序的名称。除非能找到一个名为`Command`的内链驱动，否则`Command`会在Erlang工作空间之外运行。
- `{fd, In, Out}`
  允许一个Erlang进程访问Erlang使用的任何当前打开文件描述符。文件描述符`In`可以用作标准输入，文件描述符`Out`可以用作标准输出。

`Opt`是下列选项中的一个。

- `{packet, N}`
  数据包（packet）前面有`N`（1、2或4）个字节的长度计数。
- `stream`
  发送消息时不带数据包长度信息。应用程序必须知道如何处理这些数据包。
- `{line, Max}`
  发送消息时使用一次一行的形式。如果有一行超过了`Max`字节，就会在`Max`字节处被拆分。
- `{cd, Dir}`
  只适用于`{spawn, Command}`选项。外部程序从`Dir`里启动。
- `{env, Env}`
  只适用于`{spawn, Command}`选项。外部程序的环境通过`Env`列表里的环境变量进行扩展。`Env`列表由若干个`{VarName, Value}`对组成，其中`VarName`和`Value`是字符串。

这不是open_port的完整参数清单。在erlang模块的手册页里能找到各个参数的准确细节。

下列消息可以被发往端口。请注意，所有这些消息里的PidC都是相连进程的PID。

- Port ! {PidC, {command, Data}}
  向端口发送Data（一个I/O列表）。
- Port ! {PidC, {connect, Pid1}}
  把相连进程的PID从PidC改为Pid1。
- Port ! {PidC, close}
  关闭端口。
  相连进程可以用以下方式从外部程序接收消息。

```
receive
    {Port, {data, Data}} ->
       ... 数据从外部进程进来 ...
```

在接下来的几节里，我们将建立从Erlang到一个简单C程序的接口。我们特意把这个C程序写得简短，目的是不让它干扰我们对建立接口细节的关注。

15.2　用端口建立外部 C 程序接口

我们将从一些简单的C代码开始。example1.c包含了两个函数。第一个函数计算两个整数之和，第二个函数计算参数的两倍是多少。

ports/example1.c

```
int sum(int x, int y){
  return x+y;
}

int twice(int x){
  return 2*x;
}
```

我们的最终目的是从Erlang里调用这些方法。希望能像这样调用它们：

```
X1 = example1:sum(12,23),
Y1 = example1:twice(10),
```

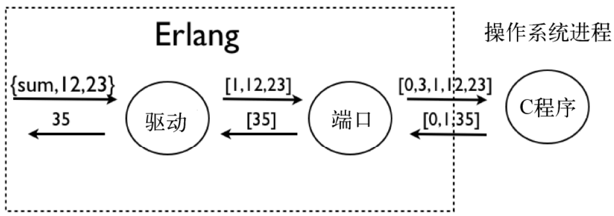
对用户而言，example1是一个Erlang模块，因此所有与C程序接口有关的细节都应该隐藏在example1模块内部。

要实现它，需要把sum(12,23)和twice(10)这样的函数调用转变成字节序列，通过端口发送给外部程序。端口给字节序列加上长度信息，然后把结果发给外部程序。当外部程序回复时，端口接收回复，并把结果发给与端口相连的进程。

所用的协议非常简单。

- 所有数据包都以2字节的长度代码（Len）开头，后接Len字节的数据。这个包头会被端口自动添加，因为打开端口时设置了参数{packet,2}。
- 把sum(N, M)调用编码成字节序列[1,N,M]。
- 把twice(N)调用编码成字节序列[2,N]。
- 参数和返回值都被假定为1字节长。

外部C程序和Erlang程序都必须遵循这一协议。下图演示了调用example1:sum(12,23)之后所发生的事。它展示了端口是如何链接外部C程序的。



整个过程如下。

(1) 驱动把函数调用sum(12,23)编码成字节序列[1,12,23]，然后向端口发送{self(), {command, [1,12,23]}}消息。

(2) 端口驱动给这个消息加上2字节的长度包头，然后把字节序列0,3,1,12,23发给外部程序。

(3) 外部程序从标准输入里读取这5个字节，调用sum函数，然后把字节序列0,1,35写入标准输出。

前两个字节包含了数据包的长度。接下来是35这个结果，它的长度是1个字节。

(4) 端口驱动移除长度包头，然后向相连进程发送一个{Port, {data, [35]}}消息。

(5) 相连进程解码这个消息，然后把结果返回给调用程序。

现在必须在接口的两侧编写遵循这一协议的程序。

15.2.1　C程序

C程序有三个文件。

- example1.c：包含了我们想要调用的函数（之前已经见过它了）。
- example1_driver.c：管理字节流协议并调用example1.c里的方法。
- erl_comm.c：带有读取和写入内存缓冲区的方法。

1. example1_driver.c

这段代码有一个循环，它会从标准输入读取命令，调用应用程序的方法，然后把结果写入标准输出。请注意，如果想要调试这个程序，可以让它写入stderr。代码里有一行注释掉的fprintf语句展示了该怎么做。

ports/example1_driver.c

```
#include <stdio.h>
#include <stdlib.h>
typedef unsigned char byte;

int read_cmd(byte *buff);
int write_cmd(byte *buff, int len);
int sum(int x, int y);
int twice(int x);

int main() {
  int fn, arg1, arg2, result;
  byte buff[100];

  while (read_cmd(buff) > 0) {
    fn = buff[0];

    if (fn == 1) {
      arg1 = buff[1];
      arg2 = buff[2];

      /* 调试，可以打印到stderr来调试
         fprintf(stderr,"calling sum %i %i\n",arg1,arg2); */
      result = sum(arg1, arg2);
    } else if (fn == 2) {
      arg1 = buff[1];
      result = twice(arg1);
    } else {
      /* 碰到未知函数就直接退出 */
      exit(EXIT_FAILURE);
    }
    buff[0] = result;
    write_cmd(buff, 1);
  }
}
```

2. erl_comm.c

最后是从标准输入和输出里读取和写入数据的代码。这段代码允许数据出现可能的分块。

ports/erl_comm.c

```
/* erl_comm.c */
#include <unistd.h>
typedef unsigned char byte;

int read_cmd(byte *buf);
int write_cmd(byte *buf, int len);
int read_exact(byte *buf, int len);
int write_exact(byte *buf, int len);
```

```
int read_cmd(byte *buf)
{
  int len;
  if (read_exact(buf, 2) != 2)
    return(-1);
  len = (buf[0] << 8) | buf[1];
  return read_exact(buf, len);
}
int write_cmd(byte *buf, int len)
{
  byte li;
  li = (len >> 8) & 0xff;
  write_exact(&li, 1);
  li = len & 0xff;
  write_exact(&li, 1);
  return write_exact(buf, len);
}
int read_exact(byte *buf, int len)
{
  int i, got=0;
  do {
    if ((i = read(0, buf+got, len-got)) <= 0)
      return(i);
    got += i;
  } while (got<len);
  return(len);
}
int write_exact(byte *buf, int len)
{
  int i, wrote = 0;
  do {
    if ((i = write(1, buf+wrote, len-wrote)) <= 0)
      return (i);
    wrote += i;
  } while (wrote<len);
  return (len);
}
```

这段代码专门用于处理带有2字节长度包头的数据，因此它与提供给端口驱动程序的{packet, 2}选项匹配。

15.2.2　Erlang程序

端口的Erlang一侧由下面这个程序驱动：

ports/example1.erl

```
-module(example1).
-export([start/0, stop/0]).
```

```
-export([twice/1, sum/2]).

start() ->
    register(example1,
             spawn(fun() ->
                     process_flag(trap_exit, true),
                     Port = open_port({spawn, "./example1"}, [{packet, 2}]),
                     loop(Port)
                   end)).

stop() ->
    ?MODULE ! stop.
twice(X) -> call_port({twice, X}).
sum(X,Y) -> call_port({sum, X, Y}).
call_port(Msg) ->
    ?MODULE ! {call, self(), Msg},
    receive
        {?MODULE, Result} ->
            Result
    end.

loop(Port) ->
    receive
        {call, Caller, Msg} ->
            Port ! {self(), {command, encode(Msg)}},
            receive
                {Port, {data, Data}} ->
                    Caller ! {?MODULE, decode(Data)}
            end,
            loop(Port);
        stop ->
            Port ! {self(), close},
            receive
                {Port, closed} ->
                    exit(normal)
            end;
        {'EXIT', Port, Reason} ->
            exit({port_terminated, Reason})
    end.

encode({sum, X, Y}) -> [1, X, Y];
encode({twice, X})  -> [2, X].

decode([Int]) -> Int.
```

这段代码遵循一个相当标准的模式。我们在start/0里创建了一个名为example1的注册进程（服务器）。call_port/1实现了对服务器的远程过程调用。twice/1和sum/2是接口方法，它们必须被导出，会对服务器发起远程过程调用。我们在loop/1里编码了对外部程序的请求，并将来自外部程序的返回值做了适当处理。

程序已经编写完成。现在需要的就是一个构建程序的makefile。

15.2.3 编译和链接端口程序

这个makefile会编译并链接本章描述的端口驱动和内链驱动程序，以及所有相关的Erlang代码。这个makefile只在Mac OS X“美洲狮”系统上测试过，其他操作系统需要进行修改。它还包含一个小型测试程序，每次代码重建时都会运行。

ports/Makefile.mac

```
.SUFFIXES: .erl .beam .yrl

.erl.beam:
	erlc -W $<

MODS = example1 example1_lid unit_test

all:	 ${MODS:%=%.beam} example1 example1_drv.so
	@erl -noshell -s unit_test start
example1:  example1.c erl_comm.c example1_driver.c
	gcc -o example1 example1.c erl_comm.c example1_driver.c
example1_drv.so: example1_lid.c example1.c
	gcc -arch i386 -I /usr/local/lib/erlang/usr/include\
	    -o example1_drv.so -fPIC -bundle -flat_namespace -undefined suppress\
	    example1.c example1_lid.c
clean:
	rm example1 example1_drv.so *.beam
```

15.2.4 运行程序

现在可以运行程序了。

```
1> example1:start().
true
2> example1:sum(45, 32).
77
4> example1:twice(10).
20
...
```

这样就完成了我们的第一个范例端口程序。这个程序实现的端口协议是Erlang与外部世界通信的主要做法。

进入下一个话题之前，请注意以下几点。

- 示例程序没有尝试去统一Erlang和C里的整数概念。只是假定Erlang和C里的整数都是一个字节，同时忽略了所有的精度和符号问题。在现实程序里，必须认真考虑相关参数的准确类型和精度。事实上，这可能是相当困难的，因为Erlang很轻松地就能管理任意大小的

整数，而C这样的语言对整数的精度和其他方面都有着严格的要求。

- 必须首先启动负责接口的驱动才能运行Erlang函数（也就是说，必须有程序先执行`example1:start()`，然后才能运行程序）。我们希望能在系统启动时自动完成这项工作。这是完全可以实现的，但需要有一定的系统启动和停止知识。我们将在23.7节中处理这件事。

15.3 在 Erlang 里调用 shell 脚本

假设想要在Erlang里调用一个shell脚本。要做到这一点，可以使用库函数`os:cmd(Str)`。它会运行字符串`Str`里的命令并捕捉结果。这里有一个使用`ifconfig`命令的例子：

```
1> os:cmd("ifconfig").
"lo0: flags=8049<UP,LOOPBACK,RUNNING,MULTICAST> mtu 16384\n\t...
```

这个结果需要先解析才能提取出我们感兴趣的信息。

15.4 高级接口技术

除了之前讨论的那些技术，建立Erlang到外部程序的接口还有一些额外的技术可用。

接下来介绍的这些技术正在不断改进，变化之快往往超过Erlang自身的演变速度。出于这个原因，这里不会详细描述它们。我们把描述转移到了在线存档里，这样它们就能得到更及时的更新。

- 内链驱动

  这些程序和之前讨论的端口驱动遵循相同的协议，唯一的区别是它们的驱动代码被链接到Erlang内核中，因此会在Erlang的操作系统主进程内运行。要构建一个内链驱动，就必须添加少量代码来初始化它，驱动本身必须被编译和链接到Erlang虚拟机上。

  `git://github.com/erlang/linked_in_drivers.git`里有最新的内链驱动范例，还介绍了如何在各种操作系统上编译它们。

- NIF

  NIF是指原生实现函数（Natively Implemented Function）。这些函数是用C（或其他能编译成本地代码的语言）编写的，并且被链接到Erlang虚拟机中。NIF直接将参数传递到Erlang进程的栈上，还能直接访问所有的Erlang内部数据结构。

  `git://github.com/erlang/nifs.git`提供了NIF的范例和最新信息。

- C-node

  C-node是用C实现的节点，它们遵循Erlang分布式协议。一个“真正的”分布式Erlang节点不仅能够与C-node通信，还会把它当作一个Erlang节点（前提是它不在C-node上做一些花哨的事情，比如发送Erlang代码让它执行）。

  http://www.erlang.org/doc/tutorial/introduction.html里的互操作性教程对C-node做了介绍。

现在我们已经了解了如何设立Erlang对外部世界的接口。接下来的几章将介绍如何在Erlang里访问文件和套接字。

15.5　练习

(1) 下载之前给出的端口驱动代码，然后在你的系统上测试它。

(2) 打开`git://github.com/erlang/linked_in_drivers.git`，下载内链驱动的代码并在你的系统上测试。这里的难点是找到编译和链接代码的正确命令。如果不能完成这个练习，可以去Erlang邮件列表寻求帮助。

(3) 看看能否找到一个操作系统命令，用它查看你的计算机使用的是哪种CPU。如果能找到这样的命令，请编写一个函数来返回你的CPU类型，做法是用`os:cmd`函数调用这个操作系统命令。

第 16 章

文件编程

16

在本章，我们将来看一些最常用的文件操作函数。标准版Erlang拥有大量的文件处理函数，我们将把重点放在其中的一小部分上。我编写的大多数程序都用到了它们，同样，你也会频繁地使用它们。还会看一些编写高效文件处理代码的技巧示例，并简要介绍一些不太常用的文件操作，让你知道它们的存在。如果想了解这些冷门技巧的更多细节，请参阅手册页。

我们将把重点放在以下领域：

- 概述用于操作文件的主要模块；
- 读取文件的不同方法；
- 写入文件的不同方法；
- 目录操作；
- 查找文件的信息 。

16.1　操作文件的模块

用于文件操作的函数被归为四个模块。

- `file`

  它包含打开、关闭、读取和写入文件的方法，还有列出目录，等等。表7简要展示了一些最常用的`file`函数。要了解所有的细节，请参阅`file`模块的手册页。

- `filename`

  这个模块里的方法能够以跨平台的方式操作文件名，这样就能在许多不同的操作系统上运行相同的代码了。

- `filelib`

  这个模块是`file`的扩展。它包含的许多工具函数能够列出文件、检查文件类型，等等。其中大多数都是使用`file`里的函数编写的。

- `io`

  这个模块有一些操作已打开文件的方法。它包含的方法能够解析文件里的数据，或者把格式化数据写入文件。

16.2 读取文件的几种方法

让我们来看看读取文件方面有什么选择。首先将编写5个小程序，它们会打开一个文件，然后用多种方式读取数据。

文件的内容仅仅是一个字节序列。它们是否有意义取决于对这些字节的解读。

为了演示这一点，我们将在所有示例里使用同一个输入文件。它实际上包含一个由各种Erlang数据类型组成的序列，根据打开和读取文件方式的不同，里面的内容可以解读成一个Erlang数据类型序列，一个文本行序列，或者没有特定意义的原始二进制数据块。

这里是文件里的原始数据：

data1.dat

```
{person, "joe", "armstrong",
        [{occupation, programmer},
         {favoriteLanguage, erlang}]}.

{cat, {name, "zorro"},
      {owner, "joe"}}.
```

现在我们将用多种方法读取这个文件的各个部分。

16.2.1 读取文件里的所有数据类型

`data1.dat`包含一个由各种Erlang数据类型组成的序列，可调用`file:consult`来读取所有的数据类型，就像下面这样：

```
1> file:consult("data1.dat").
{ok,[{person,"joe",
             "armstrong",
      [{occupation,programmer},{favoriteLanguage,erlang}]},
     {cat,{name,"zorro"},{owner,"joe"}}]}
```

`file:consult(File)`假定`File`包含一个由Erlang数据类型组成的序列。如果它能读取文件里的所有数据类型，就会返回`{ok, [Term]}`，否则会返回`{error, Reason}`。

表7 文件操作摘要（file模块）

函　　数	描　　述
`change_group`	修改某个文件所属的组
`change_owner`	修改某个文件的所有者
`change_time`	修改某个文件的最后修改或访问时间
`close`	关闭某个文件
`consult`	从某个文件里读取Erlang数据类型

（续）

函　数	描　述
`copy`	复制文件内容
`del_dir`	删除某个目录
`delete`	删除某个文件
`eval`	执行某个文件里的Erlang表达式
`format_error`	返回一个描述错误原因的字符串
`get_cwd`	获取当前工作目录
`list_dir`	列出某个目录里的文件
`make_dir`	创建一个目录
`make_link`	创建指向某个文件的硬链接（hard link）
`make_symlink`	创建指向某个文件或目录的符号链接（symbolic link）
`open`	打开某个文件
`position`	设立在文件里的位置
`pread`	对文件里的某个位置进行读取
`pwrite`	对文件里的某个位置进行写入
`read`	对某个文件进行读取
`read_file`	读取整个文件
`read_file_info`	获取某个文件的信息
`read_link`	获得某个链接指向的位置
`read_link_info`	获取某个链接或文件的信息
`rename`	重命名某个文件
`script`	执行并返回某个文件里Erlang表达式的值
`set_cwd`	设置当前工作目录
`sync`	同步某个文件在内存和物理介质中的状态
`truncate`	截断某个文件
`write`	对某个文件进行写入
`write_file`	写入整个文件
`write_file_info`	修改某个文件的信息

16.2.2　分次读取文件里的数据类型

如果想从文件里一次读取一个数据类型，就要首先用`file:open`打开文件，然后用`io:read`逐个读取数据类型，直到文件末尾，最后再用`file:close`关闭文件。

这个shell会话展示了从文件里一次读取一个数据类型时会发生什么：

```
1> {ok, S} = file:open("data1.dat", read).
{ok,<0.36.0>}
2> io:read(S, '').
{ok,{person,"joe",
     "armstrong",
     [{occupation,programmer},{favoriteLanguage,erlang}]}}
3> io:read(S, '').
{ok,{cat,{name,"zorro"},{owner,"joe"}}}
4> io:read(S, '').
eof
5> file:close(S).
ok
```

此处使用的函数如下。

- `-spec file:open(File, read) -> {ok, IoDevice} | {error, Why}`
  尝试打开`File`进行读取。如果它能打开文件就会返回`{ok, IoDevice}`，否则返回`{error, Reason}`。`IoDevice`是一个用来访问文件的I/O对象。
- `-spec io:read(IoDevice, Prompt) -> {ok, Term} | {error,Why} | eof`
  从`IoDevice`读取一个Erlang数据类型`Term`。如果`IoDevice`代表一个被打开的文件，`Prompt`就会被忽略。只有用`io:read`读取标准输入时，才会用`Prompt`提供一个提示符。
- `-spec file:close(IoDevice) -> ok | {error, Why}`
  关闭`IoDevice`。

可以用这些方法实现上一节里使用过的`file:consult`。`file:consult`可以用如下方式定义：

lib_misc.erl

```
consult(File) ->
    case file:open(File, read) of
        {ok, S} ->
            Val = consult1(S),
            file:close(S),
            {ok, Val};
        {error, Why} ->
            {error, Why}
    end.
consult1(S) ->
    case io:read(S, '') of
        {ok, Term} -> [Term|consult1(S)];
        eof        -> [];
        Error      -> Error
    end.
```

不过，`file:consult`其实不是这么定义的。标准库所用的是一个改进版，有着更好的错误报告能力。

现在是查看标准库所用版本的一个好时机。如果你能理解刚才的版本，那么理解库里的代码

就很容易了。问题只有一个：我们需要找到file.erl的源代码。可使用code:which函数来找到它，该函数能定位所有已载入模块的目标代码。

```
1> code:which(file).
"/usr/local/lib/erlang/lib/kernel-2.16.1/ebin/file.beam"
```

在标准分发套装里，每个库都有两个子目录。一个名为src，包含源代码；另一个名为ebin，包含编译后的Erlang代码。因此，file.erl的源代码应该是在下面这个目录里：

/usr/local/lib/erlang/lib/kernel-2.16.1/src/file.erl

当其他路都走不通，手册页也不能提供你想知道的代码问题答案时，快速浏览一下源代码通常可以帮你找到答案。我知道这不应该发生，但我们都是人，有时候文档的确无法回答你的所有疑问。

16.2.3　分次读取文件里的行

如果把io:read改成io:get_line，就可以分次读取文件里的行。io:get_line会一直读取字符，直到遇上换行符或者文件尾。这里有一个例子：

```
1> {ok, S} = file:open("data1.dat", read).
{ok,<0.43.0>}
2> io:get_line(S, '').
"{person, \"joe\", \"armstrong\",\n"
3> io:get_line(S, '').
"\t[{occupation, programmer},\n"
4> io:get_line(S, '').
"\t {favoriteLanguage, erlang}]}.\n"
5> io:get_line(S, '').
"\n"
6> io:get_line(S, '').
"{cat, {name, \"zorro\"},\n"
7> io:get_line(S, '').
"       {owner, \"joe\"}}.\n"
8> io:get_line(S, '').
eof
9> file:close(S).
ok
```

16.2.4　读取整个文件到二进制型中

可以用file:read_file(File)把整个文件读入一个二进制型，这是一次原子操作。

```
1> file:read_file("data1.dat").
{ok,<<"{person, \"joe\", \"armstrong\""...>>}
```

如果成功，file:read_file(File)就会返回{ok, Bin}，否则返回{error, Why}。这是

到目前为止最高效的文件读取方式，我经常用它。在大多数操作里，我会把整个文件一次性读入内存，然后操作内容并一次性保存文件（用`file:write_file`）。后面会给出一个例子。

16.2.5 通过随机访问读取文件

如果想要读取的文件非常大，或者它包含某种外部定义格式的二进制数据，就可以用`raw`模式打开这个文件，然后用`file:pread`读取它的任意部分。

这里有一个例子：

```
1> {ok, S} = file:open("data1.dat", [read,binary,raw]).
{ok,{file_descriptor,prim_file,{#Port<0.106>,5}}}
2> file:pread(S, 22, 46).
{ok,<<"rong\",\n\t[{occupation, programmer},\n\t {favorite">>}
3> file:pread(S, 1, 10).
{ok,<<"person, \"j">>}
4> file:pread(S, 2, 10).
{ok,<<"erson, \"jo">>}
5> file:close(S).
ok
```

`file:pread(IoDevice, Start, Len)`会从`IoDevice`读取`Len`个字节的数据，读取起点是字节`Start`处（文件里的字节会被编号，所以文件里第一个字节的位置是0）。它会返回`{ok, Bin}`或者`{error, Why}`。

最后，用这些随机文件访问函数来编写一个工具函数，下一章将会用到它。在17.6节里，我们将开发一个简单的SHOUTcast服务器（它是一个针对所谓流媒体的服务器，在这个案例里是MP3流）。这个服务器的要求之一是能找出内嵌在MP3文件里的艺术家和曲目名。我们将在下一节实现它。

读取MP3元数据

MP3是一种二进制格式，用来保存压缩过的音频数据。MP3文件本身并不包含有关文件内容的信息。比如说，在一个包含音乐数据的MP3文件里，音频数据并不包含录制音乐的艺术家姓名。这类数据（曲目名和艺术家姓名等）以一种被称为ID3的标签块格式保存在MP3文件中。ID3标签是由一位名叫Eric Kemp的程序员发明的，用来保存描述音频文件内容的元数据。ID3格式实际上有很多种，但基于我们的目的，这里只会编写代码来访问ID3标签的两种最简单的形式，即ID3v1和ID3v1.1标签。

ID3v1标签的结构很简单：文件最后的128个字节包含了一个固定长度的标签。前三个字节包含ASCII字符`TAG`，接下来是一些固定长度的字段。整个128字节数据是按照以下方式打包的：

长　度	内　容
3	包含TAG字符的标签头
30	标题
30	艺术家

（续）

长　度	内　容
30	专辑
4	年份
30	备注
1	流派

ID3v1的标签里没有地方可以添加曲目编号。Michael Mutschler在ID3v1.1格式里建议了一种做法，把30个字节的备注字段改成下面这样：

长　度	内　容
28	评论
1	0（一个零）
1	曲目编号

很容易就能编写一个程序来尝试读取MP3文件里的ID3v1标签，然后用二进制位匹配语法来匹配这些字段。下面就是这个程序：

id3_v1.erl

```
-module(id3_v1).
-import(lists, [filter/2, map/2, reverse/1]).
-export([test/0, dir/1, read_id3_tag/1]).
test() -> dir("/home/joe/music_keep").

dir(Dir) ->
    Files = lib_find:files(Dir, "*.mp3", true),
    L1 = map(fun(I) ->
                     {I, (catch read_id3_tag(I))}
             end,  Files),
    %% L1 = [{File, Parse}]，其中Parse = error | [{Tag,Val}]。
    %% 现在必须把所有Parse = error的条目从L里移除，
    %% 可以用一次filter操作实现。
    L2 = filter(fun({_,error}) -> false;
                   (_) -> true
                end, L1),
    lib_misc:dump("mp3data", L2).

read_id3_tag(File) ->
    case file:open(File, [read,binary,raw]) of
        {ok, S} ->
            Size = filelib:file_size(File),
            {ok, B2} = file:pread(S, Size-128, 128),
            Result = parse_v1_tag(B2),
            file:close(S),
```

```
            Result;
        _Error ->
            error
    end.
parse_v1_tag(<<$T,$A,$G,
               Title:30/binary, Artist:30/binary,
               Album:30/binary, _Year:4/binary,
               _Comment:28/binary, 0:8,Track:8,_Genre:8>>) ->
    {"ID3v1.1",
     [{track,Track}, {title,trim(Title)},
      {artist,trim(Artist)}, {album, trim(Album)}]};
parse_v1_tag(<<$T,$A,$G,
               Title:30/binary, Artist:30/binary,
               Album:30/binary, _Year:4/binary,
               _Comment:30/binary,_Genre:8>>) ->
    {"ID3v1",
     [{title,trim(Title)},
      {artist,trim(Artist)}, {album, trim(Album)}]};
parse_v1_tag(_) ->
    error.

trim(Bin) ->
    list_to_binary(trim_blanks(binary_to_list(Bin))).
trim_blanks(X) -> reverse(skip_blanks_and_zero(reverse(X))).

skip_blanks_and_zero([$\s|T]) -> skip_blanks_and_zero(T);
skip_blanks_and_zero([0|T])   -> skip_blanks_and_zero(T);
skip_blanks_and_zero(X)       -> X.
```

这个程序的主入口点是`id3_v1:dir(Dir)`。所做的第一件事是调用`lib_find:find(Dir, "*.mp3", true)`（将在16.6节进行演示）来找出所有的MP3文件，它会逐级扫描`Dir`里的目录并寻找MP3文件。

找到文件之后，调用`read_id3_tag`来解析标签。解析的任务被大大简化了，因为只需用位匹配语法就能完成它。然后就可以移除给字符串定界的尾部空白和补零字符来整理艺术家与曲目名。最后，把结果转储到一个文件里，供以后使用（21.6.2节里描述了`lib_misc:dump`）。

大多数音乐文件都带有ID3v1标签，哪怕它们同时还有ID3v2、v3和v4标签也会如此。后面这些标签标准会向文件开头（偶尔也会向文件中部）添加一个不同格式的标签。标签程序经常会既添加ID3v1标签，又向文件开头添加额外的标签（它们更难以读取）。从我们的目的出发，这里只关心包含有效ID3v1和ID3v1.1标签的文件。

了解如何读取文件之后，现在可以来看看各种写入文件的方式了。

16.3 写入文件的各种方式

写入文件所涉及的操作和读取文件基本相同。我们来了解一下。

16.3.1　把数据列表写入文件

假设想要创建一个能用file:consult读取的文件。标准库里实际上并没有这样的函数，所以我们将自己编写它。不妨把这个函数称为unconsult。

lib_misc.erl

```
unconsult(File, L) ->
    {ok, S} = file:open(File, write),
    lists:foreach(fun(X) -> io:format(S, "~p.~n",[X]) end, L),
    file:close(S).
```

可以在shell里运行它来创建一个名为 test1.dat的文件。

```
1> lib_misc:unconsult("test1.dat",
                      [{cats,["zorrow","daisy"]},
                       {weather,snowing}]).
ok
```

我们来检查一下能不能用。

```
2> file:consult("test1.dat").
{ok,[{cats,["zorrow","daisy"]},{weather,snowing}]}
```

unconsult以write模式打开文件，然后调用io:format(S, "~p.~n", [X])来把数据类型写入文件。

io:format承担了创建格式化输出的重任。要生成格式化输出，我们会做以下调用。

```
-spec io:format(IoDevice, Format, Args) -> ok
```

其中ioDevice是一个I/O对象（必须以write模式打开），Format是一个包含格式代码的字符串，Args是待输出的项目列表。

Args里的每一项都必须对应格式字符串里的某个格式命令。格式命令以一个波浪字符（~）开头。这里有一些最常用的格式命令。

- ~n输出一个换行符。~n很智能，会输出一个符合平台标准的换行符。比如说，~n在Unix机器上会把ASCII（10）写入输出流，在Windows机器上则会把回车换行ASCII（13, 10）写入输出流。
- ~p把参数打印为美观的形式。
- ~s参数是一个字符串、I/O列表或原子，打印时不带引号。
- ~w用标准语法输出数据。它被用于输出各种Erlang数据类型。

格式字符串大概有几亿个参数，一个正常思维的人是记不住的。可以在io模块的手册页里找到完整的参数清单。

我只能记住~p、~s和~n。如果你从它们开始，就不会遇到太多问题。

悄悄话

我说谎了。你需要的很可能不止~p、~s和~n。这里有一些例子：

```
Format                          Result
======                          ======
io:format("|~10s|",["abc"])     |       abc|
io:format("|~-10s|",["abc"])    |abc       |
io:format("|~10.3.+s|",["abc"]) |+++++++abc|
io:format("|~-10.10.+s|",["abc"]) |abc+++++++|
io:format("|~10.7.+s|",["abc"]) |+++abc++++|
```

16.3.2 把各行写入文件

下面这些命令和之前的例子很像，我们只是使用了不同的格式命令。

```
1> {ok, S} = file:open("test2.dat", write).
{ok,<0.62.0>}
2> io:format(S, "~s~n", ["Hello readers"]).
ok
3> io:format(S, "~w~n", [123]).
ok
4> io:format(S, "~s~n", ["that's it"]).
ok
5> file:close(S).
ok
```

这样就创建出了一个名为test2.dat的文件，它的内容如下：

```
Hello readers
123
that's it
```

16.3.3 一次性写入整个文件

这是最高效的写入文件方式。file:write_file(File, IO)会把IO里的数据（一个I/O列表）写入File。（I/O列表是一个元素为I/O列表、二进制型或0到255整数的列表。I/O列表在输出时会被自动“扁平化”，意思是所有的列表括号都会被移除。）这种方式极其高效，也是我经常用的。下一节里的程序对此做了演示。

列出文件里的URL

编写一个名为urls2htmlFile(L, File)的简单函数，它会接受一个URL列表L并生成HTML文件，里面的URL都显示为可点击链接。这样就可以采用单次I/O操作写入整个文件的技巧。我们将在scavenge_urls模块里编写程序。首先是程序的头部信息：

scavenge_urls.erl

```
-module(scavenge_urls).
-export([urls2htmlFile/2, bin2urls/1]).
-import(lists, [reverse/1, reverse/2, map/2]).
```

这个程序有两个入口点。urls2htmlFile(Urls, File)接受一个URL列表并创建HTML文

件，在文件里为每个URL各创建一个可点击的链接。bin2urls(Bin)遍历一个二进制型，然后返回一个包含该二进制型内所有URL的列表。urls2htmlFile的代码如下：

scavenge_urls.erl

```
urls2htmlFile(Urls, File) ->
    file:write_file(File, urls2html(Urls)).

bin2urls(Bin) ->  gather_urls(binary_to_list(Bin), []).

urls2html(Urls) -> [h1("Urls"),make_list(Urls)].

h1(Title) -> ["<h1>", Title, "</h1>\n"].

make_list(L) ->
    ["<ul>\n",
     map(fun(I) -> ["<li>",I,"</li>\n"] end, L),
     "</ul>\n"].
```

这段代码返回一个由字符组成的I/O列表。请注意我们没有尝试扁平化这个列表（这么做效率很低）。我们生成一个由字符组成的深层列表，然后把它直接扔给输出方法。用file:write_file把这个I/O列表写入文件时，I/O系统会自动扁平化列表（也就是说，它只会输出列表里的字符，忽略列表括号）。最后，从二进制型里提取URL的代码如下：

scavenge_urls.erl

```
gather_urls("<a href" ++ T, L) ->
    {Url, T1} = collect_url_body(T, reverse("<a href")),
    gather_urls(T1, [Url|L]);
gather_urls([_|T], L) ->
    gather_urls(T, L);
gather_urls([], L) ->
    L.

collect_url_body("</a>" ++ T, L) -> {reverse(L, "</a>"), T};
collect_url_body([H|T], L)       -> collect_url_body(T, [H|L]);
collect_url_body([], _)          -> {[],[]}.
```

要运行它，需要有一些数据来解析。输入数据（一个二进制型）是HTML网页的内容，所以需要找一张HTML网页来操作。我们将通过socket_examples:nano_get_url（参见17.1.1节）来实现。

我们将在shell里分几步完成它。

```
1> B = socket_examples:nano_get_url("www.erlang.org"),
   L = scavenge_urls:bin2urls(B),
   scavenge_urls:urls2htmlFile(L, "gathered.html").
ok
```

这样就生成了gathered.html文件。

gathered.html

```
<h1>Urls</h1>
<ul>
<li><a href="old_news.html">Older news.....</a></li>
<li><a href="http://www.erlang-consulting.com/training_fs.html">here</a></li>
<li><a href="project/megaco/">Megaco home</a></li>
<li><a href="EPLICENSE">Erlang Public License (EPL)</a></li>
<li><a href="user.html#smtp_client-1.0">smtp_client-1.0</a></li>
<li><a href="download-stats/">download statistics graphs</a></li>
<li><a href="project/test_server">Erlang/OTP Test
Server</a></li>
<li><a href="http://www.erlang.se/euc/06/">proceedings</a></li>
<li><a href="/doc/doc-5.5.2/doc/highlights.html">
       Read more in the release highlights.
</a></li>
<li><a href="index.html"><img src="images/erlang.gif"
    border="0" alt="Home"></a></li>
</ul>
```

16.3.4 写入随机访问文件

在随机访问模式下写入某个文件和读取它很相似。首先，必须用write模式打开这个文件。接下来，用file:pwrite(IoDev, Position, Bin)写入文件。

这里有一个例子：

```
1> {ok, S} = file:open("some_filename_here", [raw,write,binary]).
{ok, ...}
2> file:pwrite(S, 10, <<"new">>).
ok
3> file:close(S).
ok
```

它从文件内偏移量为10的位置开始写入字符串new，覆盖了原有内容。

16.4 目录和文件操作

file里有三个操作目录的函数。list_dir(Dir)用来生成一个Dir里的文件列表，make_dir(Dir)创建一个新目录，del_dir(Dir)删除一个目录。

如果在本书所用的代码目录里运行list_dir，就会见到类似下面这样的输出：

```
1> cd("/home/joe/book/erlang/Book/code").
/home/joe/book/erlang/Book/code
ok
2> file:list_dir(".").
{ok,["id3_v1.erl~",
```

```
"update_binary_file.beam",
"benchmark_assoc.beam",
"id3_v1.erl",
"scavenge_urls.beam",
"benchmark_mk_assoc.beam",
"benchmark_mk_assoc.erl",
"id3_v1.beam",
"assoc_bench.beam",
"lib_misc.beam",
"benchmark_assoc.erl",
"update_binary_file.erl",
"foo.dets",
"big.tmp",
...
```

请注意，这里列出的文件没有特定的顺序，不会告诉你它们是文件还是目录，也没有文件大小等信息。

要了解目录列表里各个文件的更多信息，我们会使用`file:read_file_info`，这正是下一节的主题。

16.4.1　查找文件信息

要查找文件F的信息，我们会调用`file:read_file_info(F)`。如果F是一个合法的文件或目录名，它就会返回`{ok, Info}`。`Info`是一个`#file_info`类型的记录，此类型的定义如下：

```
-record(file_info,
  {size,          % Size of file in bytes.
   type,          % Atom: device, directory, regular,
                  % or other.
   access,        % Atom: read, write, read_write, or none.
   atime,         % The local time the file was last read:
                  % {{Year, Mon, Day}, {Hour, Min, Sec}}.
   mtime,         % The local time the file was last written.
   ctime,         % The interpretation of this time field
                  % is dependent on operating system.
                  % On Unix it is the last time the file or
                  % or the inode was changed.  On Windows,
                  % it is the creation time.
   mode,          % Integer: File permissions.  On Windows,
                  % the owner permissions will be duplicated
                  % for group and user.
   links,         % Number of links to the file (1 if the
                  % filesystem doesn't support links).
   ...
   }).
```

注意　mode和access字段有重叠。可以用mode一次性设置多个文件的属性，也可以用更简单的access操作。

要查找某个文件的大小和类型，就会调用read_file_info，就像下面这个例子（请注意，我们必须包含file.hrl，因为它内含#file_info记录的定义）：

lib_misc.erl

```
-include_lib("kernel/include/file.hrl").
file_size_and_type(File) ->
    case file:read_file_info(File) of
        {ok, Facts} ->
            {Facts#file_info.type, Facts#file_info.size};
        _ ->
            error
    end.
```

现在可以增强list_file返回的目录清单了，方法是在ls()函数里添加文件信息，就像下面这样：

lib_misc.erl

```
ls(Dir) ->
    {ok, L} = file:list_dir(Dir),
    lists:map(fun(I) -> {I, file_size_and_type(I)} end, lists:sort(L)).
```

现在，当我们列出文件时，它们会按顺序排列并包含额外的有用信息。

```
1> lib_misc:ls(".").
[{"Makefile",{regular,1244}},
 {"README",{regular,1583}},
 {"abc.erl",{regular,105}},
 {"alloc_test.erl",{regular,303}},
 ...
 {"socket_dist",{directory,4096}},
 ...
```

为了方便起见，filelib模块导出了一些小方法，比如file_size(File)和is_dir(X)。它们只不过是file:read_file_info的接口。如果只想获得文件大小，更方便的做法是调用filelib:file_size，而不是调用file:read_file_info然后解包#file_info记录里的元素。

16.4.2 复制和删除文件

file:copy(Source, Destination)会把文件Source复制到Destination里。

file:delete(File)会删除File。

16.5 其他信息

到目前为止，我们已经介绍了我日常用来操作文件的大多数函数。我们不打算对下列主题进行深入讨论，你可以在手册页里找到更多细节。

- 文件模式
  用file:open打开文件时，我们会以某个或某一组模式来打开它。事实上，模式的数量比我们想象的要多得多。举个例子，读取和写入gzip压缩文件时可以使用compressed这个模式标记。手册页里有完整的清单。
- 修改时间、用户组、符号链接
  可以用file里的一些方法来设置它们。
- 错误代码
  我曾经泛泛地说过所有错误都是{error, Why}这种形式。事实上，Why是一个原子（比如用enoent表示文件不存在，等等）。错误代码的数量其实有很多，手册页里对它们都进行了描述。
- filename
  filename模块里有一些很有用的方法，比如拆分目录里的完整文件名来获得文件扩展名，以及用各个组成部分重建文件名，等等。所有这些都是以跨平台的方式实现的。
- filelib
  filelib模块有一些小方法能给我们减少一点工作量。举个例子，filelib:ensure_dir(Name)会确保给定文件或目录名Name的所有上级目录都存在，必要的话会尝试创建它们。

16.6　一个查找工具函数

在最后一个例子里，我们将用file:list_dir和file:read_file_info来编写一个通用型“查找”工具。

这个模块的主要入口点如下：

```
lib_find:files(Dir, RegExp, Recursive, Fun, Acc0)
```

里面的参数如下。

- Dir
  这个目录名是文件搜索的起点。
- RegExp
  这是一个shell风格的正则表达式，用于测试我们找到的文件。如果遇到的文件匹配这个正则表达式，就会调用Fun(File, Acc)，其中File是匹配正则表达式的文件名。
- Recursive = true | false
  这个标记决定了搜索是否应该层层深入当前搜索目录的子目录。
- Fun(File, AccIn) -> AccOut
  如果regExp匹配File，这个函数就会被应用到File上。Acc是一个初始值为Acc0的归集器。Fun在每次调用后必须返回一个新的归集值，这个值会在下次调用Fun时传递给它。归集器的最终值就是lib_find:files/5的返回值。

可以向lib_find:files/5传递任何我们喜欢的函数。举个例子，可以用下面这个函数创建一个文件列表，并给它传递一个空列表作为初始值：

```
fun(File, Acc) -> [File|Acc] end
```

模块入口点lib_find:files(Dir, ShellRegExp, Flag)为此程序的最常用功能提供了一个简化的入口点。这里的ShellRegExp是一个shell风格的通配符模式，比完整形式的正则表达式更容易编写。

作为这种简化调用形式的一个例子，下面的调用：

```
lib_find:files(Dir, "*.erl", true)
```

会逐层查找Dir目录下的所有Erlang文件。如果最后那个参数是false，就只会查找Dir目录里的Erlang文件，而不会深入子目录。

最后是代码：

lib_find.erl

```
-module(lib_find).
-export([files/3, files/5]).
-import(lists, [reverse/1]).

-include_lib("kernel/include/file.hrl").

files(Dir, Re, Flag) ->
    Re1 = xmerl_regexp:sh_to_awk(Re),
    reverse(files(Dir, Re1, Flag, fun(File, Acc) ->[File|Acc] end, [])).

files(Dir, Reg, Recursive, Fun, Acc) ->
    case file:list_dir(Dir) of
        {ok, Files} -> find_files(Files, Dir, Reg, Recursive, Fun, Acc);
        {error, _}  -> Acc
    end.

find_files([File|T], Dir, Reg, Recursive, Fun, Acc0) ->
    FullName = filename:join([Dir,File]),
    case file_type(FullName) of
        regular ->
            case re:run(FullName, Reg, [{capture,none}]) of
                match  ->
                    Acc = Fun(FullName, Acc0),
                    find_files(T, Dir, Reg, Recursive, Fun, Acc);
                nomatch ->
                    find_files(T, Dir, Reg, Recursive, Fun, Acc0)
            end;
        directory ->
            case Recursive of
                true ->
```

```
                    Acc1 = files(FullName, Reg, Recursive, Fun, Acc0),
                    find_files(T, Dir, Reg, Recursive, Fun, Acc1);
                false ->
                    find_files(T, Dir, Reg, Recursive, Fun, Acc0)
            end;
        error ->
            find_files(T, Dir, Reg, Recursive, Fun, Acc0)
    end;
find_files([], _, _, _, _, A) ->
    A.

file_type(File) ->
    case file:read_file_info(File) of
        {ok, Facts} ->
            case Facts#file_info.type of
                regular   -> regular;
                directory -> directory;
                _         -> error
            end;
        _ ->
            error
    end.
```

这就是查找文件的方法。你可能注意到这个程序都是顺序的。要加速它，可以用多个并行进程来实现并行处理。我不会在这里做这件事，你可以思考一下怎么做。

把Erlang里的文件访问和并发结合在一起，我们就有了一个解决复杂问题的强力工具。如果想要并行分析大量的文件，就可以分裂出许多进程，让它们各自分析一个文件。我们唯一需要注意的问题就是必须确保两个进程不会同时读取和写入同一个文件。

实现高效文件操作的方法是在一次操作里进行文件的读取和写入，并在写入文件前先创建I/O列表。

16.7　练习

(1) 编译Erlang文件`X.erl`后会生成一个`X.beam`文件（如果编译成功的话）。编写一个程序来检查某个Erlang模块是否需要重新编译。做法是比较相关Erlang文件和beam文件的最后修改时间戳。

(2) 编写一个程序来计算某个小文件的MD5校验和，做法是用内置函数`erlang:md5/1`来计算文件数据的MD5校验和（有关这个内置函数的详情请参阅Erlang手册页）。

(3) 对一个大文件（比如几百MB）重复前面的练习。这次分小块读取该文件，并用`erlang:md5_init`、`erlang:md5_update`和`erlang:md5_final`计算该文件的MD5校验和。

(4) 用`lib_find`模块查找计算机里的所有`.jpg`文件。计算每一个文件的MD5校验和，然后比较校验和来看看是否存在两张相同的图片。

(5) 编写一种缓存机制，让它计算文件的MD5校验和，然后把结果和文件的最后修改时间一

起保存在缓存里。当想要某个文件的MD5值时就检查缓存，看看是否已经计算过，如果文件的最后修改时间有变化就重新计算它。

(6) 一条推特刚好是140字节长。编写一个名为`twit_store.erl`的随机访问式推特存储模块，并导出下列函数：`init(K)`分配`K`条推特的空间。`store(N, Buf)`在存储区里保存第`N`（范围是`1`至`K`）条推特的数据`Buf`（一个140字节的二进制型）。`fetch(N)`取出第`N`条推特的数据。

第 17 章 套接字编程

我编写过的大多数有趣的程序都直接或间接涉及套接字（socket）。套接字编程很有乐趣，因为它能让应用程序与互联网上的其他机器交互，这比只进行本地操作有更大的空间。

套接字是一种通信信道，让不同的机器能用互联网协议（Internet Protocol，简称IP）在网上通信。在这一章里，我们将把重点放在两个核心互联网协议上：*传输控制协议*（Transmission Control Protocol，简称TCP）和*用户数据报协议*（User Datagram Protocol，简称UDP）。

UDP能让应用程序相互发送简短的消息（称为*数据报*），但是并不保证这些消息能成功到达。它们也可能会不按照发送顺序到达。而TCP能提供可靠的字节流，只要连接存在就会按顺序到达。用TCP发送数据的额外开销比用UDP发送数据更大。可以选择可靠但更慢的信道（TCP），也可以选择更快但不可靠的信道（UDP）。

套接字编程有两个主要的库：`gen_tcp`用于编写TCP应用程序，`gen_udp`用于编写UDP应用程序。

本章将带我们了解如何用TCP和UDP套接字来编写客户端与服务器。我们将讨论各种不同的服务器形式（并行式、顺序式、阻塞式和非阻塞式），并看看如何编写流量整形（traffic-shaping）应用程序来控制进入程序的数据流。

17.1 使用 TCP

我们将以一个简单的TCP程序作为这场套接字编程历险的起点，它将从某个服务器上获取信息。在此之后，将编写一个简单的顺序TCP服务器，并展示如何让它并行化来处理多个并行会话。

17.1.1 从服务器获取数据

首先，编写一个名为`nano_get_url/0`的小函数，它用一个TCP套接字来从`http://www.google.com`获取一张HTML网页。

socket_examples.erl

```
nano_get_url() ->
    nano_get_url("www.google.com").

nano_get_url(Host) ->
```

```
❶       {ok,Socket} = gen_tcp:connect(Host,80,[binary, {packet, 0}]),
❷       ok = gen_tcp:send(Socket, "GET / HTTP/1.0\r\n\r\n"),
        receive_data(Socket, []).

    receive_data(Socket, SoFar) ->
        receive
❸           {tcp,Socket,Bin} ->
                receive_data(Socket, [Bin|SoFar]);
❹           {tcp_closed,Socket} ->
❺               list_to_binary(reverse(SoFar))
        end.
```

它的工作方式如下。

❶ 调用`gen_tcp:connect`来打开一个到`http://www.google.com` 80端口的TCP套接字。连接调用里的`binary`参数告诉系统要以“二进制”模式打开套接字，并把所有数据用二进制型传给应用程序。`{packet,0}`的意思是把未经修改的TCP数据直接传给应用程序。

❷ 调用`gen_tcp:send`，把消息`GET / HTTP/1.0\r\n\r\n`发送给套接字，然后等待回复。这个回复并不是放在一个数据包里，而是分成多个片段，一次发送一点。这些片段会被接收成为消息序列，发送给打开（或控制）套接字的进程。

❸ 收到一个`{tcp,Socket,Bin}`消息。这个元组的第三个参数是一个二进制型，原因是打开套接字时使用了二进制模式。这个消息是Web服务器发送给我们的数据片段之一。把它添加到目前已收到的片段列表中，然后等待下一个片段。

❹ 收到一个`{tcp_closed, Socket}`消息。这会在服务器完成数据发送时发生。

❺ 当所有片段都到达后，因为它们的保存顺序是错误的，所以反转它们并连接所有片段。

重新组装片段的代码是这样的：

```
receive_data(Socket, SoFar) ->
    receive
        {tcp,Socket,Bin} ->
            receive_data(Socket, [Bin|SoFar]);
        {tcp_closed,Socket} ->
            list_to_binary(reverse(SoFar))
    end.
```

这样，随着片段陆续抵达，要做的就是把它们添加到`SoFar`的列表头。当所有片段都已到达并且套接字关闭后，反转这个列表并连接所有片段。

你可能会认为更好的做法是像这样编写代码来归集片段：

```
receive_data(Socket, SoFar) ->
    receive
        {tcp,Socket,Bin} ->
            receive_data(Socket, list_to_binary([SoFar,Bin]));
        {tcp_closed,Socket} ->
            SoFar
    end.
```

这段代码并没有错，但是比原先的版本效率更低。原因是我们在后一个版本里持续将一个新的二进制型添加到缓冲区末尾，而这需要进行大量的数据复制。更好的方法是把所有片段归集到一个列表里（会是错误的顺序），然后反转整个列表，在单次操作里连接所有片段。

来测试一下这个小型HTTP客户端能不能用。

```
1> B = socket_examples:nano_get_url().
<<"HTTP/1.0 302 Found\r\nLocation: http://www.google.se/\r\n
Cache-Control: private\r\nSet-Cookie: PREF=ID=b57a2c:TM"...>>
```

注意　运行nano_get_url时，结果会是一个二进制型，这样你就能看到二进制型在Erlang shell里美化打印后是什么样子的。当你美化打印二进制型时，所有的控制字符都会显示成转义格式。而且这个二进制型是被截断的，输出内容后面的省略号（...>>）表明了这一点。如果想查看整个二进制型，可以用io:format打印它，或者用string:tokens把它分成几部分。

编写一个Web服务器

编写Web客户端或服务器这样的程序是非常有乐趣的。是的，其他人已经编写过这些程序，但如果真想知道它们是如何工作的，就可以深入本质来了解它们的工作方式。谁知道呢，也许我们的Web服务器会比现有的更出色。

要创建一个Web服务器或者其他任何实现标准互联网协议的软件，需要使用正确的工具，并且要确切知道该实现哪种协议。

在获取网页的示例代码里，打开80端口并发给它一个GET / HTTP/1.0\r\n\r\n命令。使用RFC 1945里定义的HTTP协议。互联网服务的所有主要协议都在各自的征求评议文件（Requests for Comments，简称RFC）里进行定义。所有RFC的官方网站是http://www.ietf.org（互联网工程任务组的主页）。

另一个宝贵的信息来源是包嗅探器（packet sniffer）。有了包嗅探器，就可以捕捉并分析所有进出应用程序的IP数据包。大多数包嗅探器都包含能解码并分析包内数据的软件，还能以有意义的方式呈现数据。Wireshark（以前叫Ethereal）是其中最知名也可能是最好的，下载地址是http://www.wireshark.org

有了包嗅探器的转储信息和合适的RFC作武器，就可以着手编写下一个杀手级应用程序了。

```
2> io:format("~p~n",[B]).
<<"HTTP/1.0 302 Found\r\nLocation: http://www.google.se/\r\n
  Cache-Control: private\r\nSet-Cookie: PREF=ID=b57a2c:TM"
  TM=176575171639526:LM=1175441639526:S=gkfTrK6AFkybT3;
  expires=Sun, 17-Jan-2038 19:14:07
  ... several lines omitted ...
>>
```

```
3> string:tokens(binary_to_list(B),"\r\n").
["HTTP/1.0 302 Found",
 "Location: http://www.google.se/",
 "Cache-Control: private",
 "Set-Cookie: PREF=ID=ec7f0c7234b852dece4:TM=11713424639526:
 LM=1171234639526:S=gsdertTrK6AEybT3;
 expires=Sun, 17-Jan-2038 19:14:07 GMT; path=/; domain=.google.com",
 "Content-Type: text/html",
 "Server: GWS/2.1",
 "Content-Length: 218",
 "Date: Fri, 16 Jan 2009 15:25:26 GMT",
 "Connection: Keep-Alive",
... lines omitted ...
```

请注意，响应码302不是错误，它是预期的命令响应，意思是重定向到一个新地址。另外请注意这个例子展示了套接字通信的工作方式，而且没有严格遵循HTTP协议。

这或多或少是一个Web客户端的工作方式（重点在少上，因为我们要做很多工作才能在Web浏览器里正确渲染出结果数据）。但是，之前的代码是你自己进行试验的一个良好起点。你可能会想试着修改代码来获取和保存整个网站，或者自动去读取电子邮件。可能性是无穷的。

17.1.2 一个简单的TCP服务器

我们在上一节里编写了一个简单的客户端。现在来编写一个服务器。

这个服务器会打开2345端口，然后等待一个消息。这个消息是一个二进制型，内含一个Erlang数据类型。这个数据类型是一个Erlang字符串，内含一个表达式。服务器会执行这个表达式，然后把结果发给客户端，方法是把结果写入套接字。

要编写这个程序（以及任何运行在TCP/IP上的程序），必须回答一些简单的问题。

- 数据是如何组织的？如何知道单次请求或响应包含多少数据？
- 请求或响应里的数据是如何编码和解码的？（编码数据有时被称为marshaling，即编组。解码数据有时被称为demarshaling，即解编。）

TCP套接字数据只不过是一个无差别的字节流。这些数据在传输过程中可以被打散成任意大小的片段，所以需要事先约定，这样才能知道多少数据代表一个请求或响应。

我们在Erlang里使用了一种简单的约定，即每个逻辑请求或响应前面都会有一个N（1、2或4）字节的长度计数。这就是`gen_tcp:connect`和`gen_tcp:listen`函数里参数`{packet, N}`的意思。packet这个词在这里指的是应用程序请求或响应消息的长度，而不是网络上的实际数据包。需要注意的是，客户端和服务器使用的`packet`参数必须一致。如果启动服务器时用了`{packet,2}`，客户端用了`{packet,4}`，程序就会失败。

用`{packet,N}`选项打开一个套接字后，无需担心数据碎片的问题。Erlang驱动会确保所有碎片化的数据消息首先被重组成正确的长度，然后才会传给应用程序。

下一个要关心的问题是数据*编码和解码*。我们将用最简单的方式来编码和解码消息，也就是用`term_to_binary`编码Erlang数据类型，然后用它的逆函数`binary_to_term`解码数据。

请注意，客户端与服务器通信所需的打包约定和编码规则是由两行代码实现的，第一行是在打开套接字时使用{packet,4}选项，第二行是用term_to_binary和它的逆函数来编码与解码数据。

相比HTTP或XML这些基于文本的方法，打包和编码Erlang数据的便利性给了我们很大的优势。使用Erlang内置函数term_to_binary和它的逆函数binary_to_term通常会比用XML数据的同类操作快不止一个数量级，而且发送的数据也会少很多。现在来看程序。首先，这里有一个非常简单的服务器：

socket_examples.erl

```
   start_nano_server() ->
❶      {ok, Listen} = gen_tcp:listen(2345, [binary, {packet, 4},
                                             {reuseaddr, true},
                                             {active, true}]),
❷      {ok, Socket} = gen_tcp:accept(Listen),
❸      gen_tcp:close(Listen),
       loop(Socket).
   loop(Socket) ->
       receive
           {tcp, Socket, Bin} ->
               io:format("Server received binary = ~p~n",[Bin]),
❹              Str = binary_to_term(Bin),
               io:format("Server (unpacked)  ~p~n",[Str]),
❺              Reply = lib_misc:string2value(Str),
               io:format("Server replying = ~p~n",[Reply]),
❻              gen_tcp:send(Socket, term_to_binary(Reply)),
               loop(Socket);
           {tcp_closed, Socket} ->
               io:format("Server socket closed~n")
       end.
```

它的工作方式如下。

❶ 首先，调用gen_tcp:listen来监听2345端口的连接，并设置消息的打包约定。{packet, 4}的意思是每个应用程序消息前部都有一个4字节的长度包头。然后gen_tcp:listen(..)会返回{ok, Listen}或{error, Why}，但我们只关心能够打开套接字的返回值。因此，编写如下代码：

```
{ok, Listen} = gen_tcp:listen(...),
```

这会让程序在gen_tcp:listen返回{error, ...}时抛出一个模式匹配异常错误。在成功的情况下，这个语句会绑定Listen到刚监听的套接字上。我们只能对监听端口做一件事，那就是把它用作gen_tcp:accept的参数。

❷ 现在调用gen_tcp:accept(Listen)。在这个阶段，程序会挂起并等待一个连接。当我们收到连接时，这个函数就会返回变量Socket，它绑定了可以与连接客户端通信的套接字。

❸ 在accept返回后立即调用gen_tcp:close(Listen)。这样就关闭了监听套接字，使服务

器不再接收任何新连接。这么做不会影响现有连接，只会阻止新连接。

❹ 解码输入数据（解编）。

❺ 然后执行字符串。

❻ 然后编码回复数据（编组）并把它发回套接字。

请注意，这个程序只接收单个请求。一旦程序运行结束，就不会再接收新连接了。

这个最简单的服务器演示了如何打包和编码应用程序数据。它接收一个请求，计算出回复，发送回复，然后终止。

要测试这个服务器，需要一个对应的客户端。

socket_examples.erl

```
nano_client_eval(Str) ->
    {ok, Socket} =
        gen_tcp:connect("localhost", 2345,
                        [binary, {packet, 4}]),
    ok = gen_tcp:send(Socket, term_to_binary(Str)),
    receive
        {tcp,Socket,Bin} ->
            io:format("Client received binary = ~p~n",[Bin]),
            Val = binary_to_term(Bin),
            io:format("Client result = ~p~n",[Val]),
            gen_tcp:close(Socket)
    end.
```

为了测试这些代码，我们将在同一台机器上运行客户端和服务器，这样`gen_tcp:connect`函数里的主机名就是固定的`localhost`。

请注意客户端如何调用`term_to_binary`来编码消息，以及服务器如何调用`binary_to_term`来重建消息。

要运行它，需要打开两个终端窗口，并在每个窗口里都启动一个Erlang shell。

首先，启动服务器。

```
1> socket_examples:start_nano_server().
```

我们不会在服务器窗口里看到任何输出，因为还没有什么事情发生。然后转到客户端窗口并输入以下命令：

```
1> socket_examples:nano_client_eval("list_to_tuple([2+3*4,10+20])").
```

我们应该会在服务器窗口里看到以下输出：

```
Server received binary = <<131,107,0,28,108,105,115,116,95,116,
                          111,95,116,117,112,108,101,40,91,50,
                          43,51,42,52,44,49,48,43,50,48,93,41>>
Server (unpacked)  "list_to_tuple([2+3*4,10+20])"
Server replying = {14,30}
```

在客户端窗口里，我们则会看到这些：

```
Client received binary = <<131,104,2,97,14,97,30>>
Client result = {14,30}
ok
```

最后，我们会在服务器窗口里看到这个：

```
Server socket closed
```

17.1.3 顺序和并行服务器

在上一节里，我们编写的服务器只接收一个连接就会关闭。把这段代码稍作修改，我们就能写出两种不同类型的服务器。

- 顺序服务器：一次接收一个连接
- 并行服务器：同时接收多个并行连接

原来的代码开头是这样的：

```
start_nano_server() ->
    {ok, Listen} = gen_tcp:listen(...),
    {ok, Socket} = gen_tcp:accept(Listen),
    loop(Socket).
 ...
```

我们将对它进行修改来制作两种服务器变体。

1. 顺序服务器

为了制作一个顺序服务器，我们把代码修改如下：

```
start_seq_server() ->
    {ok, Listen} = gen_tcp:listen(...),
    seq_loop(Listen).

seq_loop(Listen) ->
    {ok, Socket} = gen_tcp:accept(Listen),
    loop(Socket),
    seq_loop(Listen).

loop(..) -> %% 和之前一样
```

它的工作方式和前面的例子基本一致，但因为我们想要服务的请求不止一个，所以让监听套接字保持打开状态，不会调用`gen_tcp:close(Listen)`。另一处区别是在`loop(Socket)`完成后再次调用`seq_loop(Listen)`，让它等待下一个连接。

如果一个客户端尝试连接时服务器正忙于处理现有连接，该连接就会加入队列，直至服务器完成现有连接。如果排队的连接数量超过了监听缓冲区限制，该连接就会被拒绝。

我们只展示了启动服务器的代码。停止服务器很简单（停止并行服务器也一样），只需终止启动单个或多个服务器的进程即可。`gen_tcp`自身会连接到控制进程上，如果控制进程终止，它就会关闭套接字。

2. 并行服务器

制作并行服务器的诀窍是：每当gen_tcp:accept收到一个新连接时就立即分裂一个新进程。

```
start_parallel_server() ->
    {ok, Listen} = gen_tcp:listen(...),
    spawn(fun() -> par_connect(Listen) end).

par_connect(Listen) ->
    {ok, Socket} = gen_tcp:accept(Listen),
    spawn(fun() -> par_connect(Listen) end),
    loop(Socket).

loop(..) -> %% 和之前一样
```

这段代码类似于之前看到的顺序服务器。关键的区别在于添加了一个spawn，这就确保了我们会为每个新套接字连接创建一个并行进程。现在是比较这两者的好机会。你应该观察spawn语句的摆放位置，看看它们是如何把一个顺序服务器转变成并行服务器的。

这三个服务器全都调用了gen_tcp:listen和gen_tcp:accept，唯一的区别在于调用这些函数时是在并行程序还是在顺序程序里。

17.1.4 注意事项

需要注意以下几点。

- 创建某个套接字（通过调用gen_tcp:accept或gen_tcp:connect）的进程被称为该套接字的控制进程。所有来自套接字的消息都会被发送到控制进程。如果控制进程挂了，套接字就会被关闭。某个套接字的控制进程可以通过调用gen_tcp:controlling_process(Socket, NewPid)修改成NewPid。
- 我们的并行服务器可能会创建出几千个连接，所以可以限制最大同时连接数。实现的方法可以是维护一个计数器来统计任一时刻有多少活动连接。每当收到一个新连接时就让计数器加1，每当一个连接结束时就让它减1。可以用它来限制系统里的同时连接总数。
- 接受一个连接后，显式设置必要的套接字选项是一种很好的做法，就像这样：

```
{ok, Socket} = gen_tcp:accept(Listen),
inet:setopts(Socket, [{packet,4},binary,
                      {nodelay,true},{active, true}]),
loop(Socket)
```

- Erlang的R11B-3版开始允许多个Erlang进程对同一个监听套接字调用gen_tcp:accept/1。这让编写并行服务器变得简单了，因为你可以生成一个预先分裂好的进程池，让它们都处在gen_tcp:accept/1的等待状态。

17.2　主动和被动套接字

Erlang的套接字可以有三种打开模式：主动（active）、单次主动（active once）或被动（passive）。这是通过在`gen_tcp:connect(Address, Port, Options)`或`gen_tcp:listen(Port, Options)`的`Options`参数里加入`{active, true | false | once}`选项实现的。

如果指定`{active, true}`就会创建一个主动套接字，指定`{active, false}`则是被动套接字。`{active, once}`创建的套接字只会主动接收一个消息，接收完之后必须重新启用才能接收下一个消息。

我们将在接下来的几节里介绍这些不同类型套接字的用途。

主动和被动套接字的区别在于套接字接收到消息之后所发生的事。

- 当一个主动套接字被创建后，它会在收到数据时向控制进程发送`{tcp, Socket, Data}`消息。控制进程无法控制这些消息流。恶意的客户端可以向系统发送成千上万的消息，而它们都会被发往控制进程。控制进程无法阻止这些消息流。
- 如果一个套接字是用被动模式打开的，控制进程就必须调用`gen_tcp:recv(Socket, N)`来从这个套接字接收数据。然后它会尝试从套接字接收N个字节。如果`N = 0`，套接字就会返回所有可用的字节。在这个案例里，服务器可以通过选择何时调用`gen_tcp:recv`来控制客户端所发的消息流。

被动套接字的作用是控制通往服务器的数据流。要演示这一点，我们可以用三种方式编写服务器的消息接收循环。

- 主动消息接收（非阻塞式）
- 被动消息接收（阻塞式）
- 混合消息接收（部分阻塞式）

17.2.1　主动消息接收（非阻塞式）

第一个例子用主动模式打开一个套接字，然后从这个套接字里接收消息。

```
{ok, Listen} = gen_tcp:listen(Port, [..,{active, true}...]),
{ok, Socket} = gen_tcp:accept(Listen),
loop(Socket).

loop(Socket) ->
    receive
        {tcp, Socket, Data} ->
            ... 对数据进行操作 ...
        {tcp_closed, Socket} ->
            ...
    end.
```

这个进程无法控制通往服务器循环的消息流。如果客户端生成数据的速度快于服务器处理数据的速度，系统就会遭受数据洪流的冲击：消息缓冲区会被塞满，系统可能会崩溃或表现异常。

这种类型的服务器被称为非阻塞式服务器，因为它无法阻挡客户端。只有在确信服务器能跟上客户端的需求时才会编写非阻塞式服务器。

17.2.2　被动消息接收（阻塞式）

在这一节里，我们将编写一个阻塞式服务器。这个服务器通过设置{active, false}选项用被动模式打开套接字。这个服务器不会因为某个过激的客户端试图用过量数据冲击它而崩溃。

服务器循环里的代码会在每次想要接收数据时调用gen_tcp:recv。客户端会一直被阻塞，直到服务器调用recv为止。请注意，操作系统有自己的缓冲设置，即使没有调用recv，客户端也能在阻塞前发送少量数据。

```
{ok, Listen} = gen_tcp:listen(Port, [..,{active, false}...]),
{ok, Socket} = gen_tcp:accept(Listen),
loop(Socket).

loop(Socket) ->
    case gen_tcp:recv(Socket, N) of
        {ok, B} ->
            ... 对数据进行操作 ...
            loop(Socket);
        {error, closed}
            ...
    end.
```

17.2.3　混合消息接收（部分阻塞式）

你可能会认为对所有服务器都使用被动模式是正确的做法。只不过，当我们处于被动模式时，只能等待来自单个套接字的数据。这对于编写那些必须等待来自多个套接字数据的服务器来说毫无用处。

幸运的是，我们可以采用一种混合方式，既不是阻塞也不是非阻塞。用{active, once}选项打开套接字。套接字在这个模式下虽然是主动的，但只针对一个消息。当控制进程收到一个消息后，必须显式调用inet:setopts才能重启下一个消息的接收，在此之前系统会处于阻塞状态。这种方法集合了前两种模式的优点，它的代码如下：

```
{ok, Listen} = gen_tcp:listen(Port, [..,{active, once}...]),
{ok, Socket} = gen_tcp:accept(Listen),
loop(Socket).

loop(Socket) ->
    receive
        {tcp, Socket, Data} ->
            ... 对数据进行操作 ...
            %% 当你准备好启动下一个消息的接收时
            inet:setopts(Sock, [{active, once}]),
```

```
            loop(Socket);
        {tcp_closed, Socket} ->
            ...
    end.
```

通过使用{active, once}选项，用户可以实现高级形式的流量控制（有时被称为流量整形），从而防止服务器被过多消息淹没。

找出连接的来源

假设编写了某种在线服务器，并且发现有人持续向网站发送垃圾信息。为了尽量防止这种事发生，我们需要知道连接的来源。可以调用inet:peername(Socket)进行查看。

```
@spec inet:peername(Socket) -> {ok, {IP_Address, Port}} | {error, Why}
```

它会返回连接另一端的IP地址和端口号，这样服务器就能看到是谁发起的连接。IP_Address是一个由整数组成的元组，{N1,N2,N3,N4}代表IPv4地址，{K1,K2,K3,K4,K5,K6,K7,K8}代表IPv6地址。这里的Ni是0到255之间的整数，Ki是0到65535之间的整数。

17.3　套接字错误处理

套接字的错误处理极其简单，基本上你什么都不用做。正如我们之前所说的，每个套接字都有一个控制进程（也就是创建该套接字的进程）。如果控制进程挂了，套接字就会被自动关闭。

这意味着什么呢？举个例子，如果我们有一个客户端和一个服务器，而服务器因为程序错误挂了，那么服务器支配的套接字就会被自动关闭，同时向客户端发送一个{tcp_closed, Socket}消息。

可以用下面这个小程序来测试一下这种机制：

socket_examples.erl

```
error_test() ->
    spawn(fun() -> error_test_server() end),
    lib_misc:sleep(2000),
    {ok,Socket} = gen_tcp:connect("localhost",4321,[binary, {packet, 2}]),
    io:format("connected to:~p~n",[Socket]),
    gen_tcp:send(Socket, <<"123">>),
    receive
        Any ->
            io:format("Any=~p~n",[Any])
    end.
error_test_server() ->
    {ok, Listen} = gen_tcp:listen(4321, [binary,{packet,2}]),
    {ok, Socket} = gen_tcp:accept(Listen),
    error_test_server_loop(Socket).
```

```
error_test_server_loop(Socket) ->
    receive
        {tcp, Socket, Data} ->
            io:format("received:~p~n",[Data]),
            _ = atom_to_list(Data),
            error_test_server_loop(Socket)
    end.
```

运行它时会看到以下输出：

```
1> socket_examples:error_test().
connected to:#Port<0.152>
received:<<"123">>
=ERROR REPORT==== 30-Jan-2009::16:57:45 ===
Error in process <0.77.0> with exit value:
 {badarg,[{erlang,atom_to_list,[<<3 bytes>>]},
 {socket_examples,error_test_server_loop,1}]}
Any={tcp_closed,#Port<0.152>}
ok
```

我们分裂出一个服务器并睡眠两秒钟来让它完成启动，然后向它发送一个包含二进制型`<<"123">>`的消息。当这个消息到达服务器后，服务器尝试对二进制型`Data`计算`atom_to_list(Data)`，于是立即崩溃了。系统监视器打印出了你在shell里所见到的诊断信息。因为服务器端套接字的控制进程已经崩溃，所以（服务器端的）套接字就被自动关闭了。系统随后向客户端发送一个`{tcp_closed, Socket}`消息。

17.4 UDP

现在让我们来看看用户数据报协议（UDP）。互联网上的机器能够通过UDP相互发送被称为*数据报*（datagram）的短消息。UDP数据报是不可靠的，这就意味着如果客户端向服务器发送一串UDP数据报，它们可能会不按顺序到达，不能成功到达或者不止一次到达。但是每一个数据报只要到达，就会是完好无损的。大型数据报可以被拆分成若干个小片段，但IP协议会在传输到应用程序之前重新组合这些片段。

UDP是一种*无连接*协议，意思是客户端向服务器发送消息之前不必建立连接。这就意味着UDP非常适合那些大量客户端向服务器发送简短消息的应用程序。

用Erlang编写UDP客户端和服务器要比编写TCP程序简单得多，因为我们无需担心服务器连接的维护工作。

17.4.1 最简单的UDP服务器与客户端

首先来讨论服务器。UDP服务器的基本形式如下：

```
server(Port) ->
    {ok, Socket} = gen_udp:open(Port, [binary]),
    loop(Socket).
```

```
loop(Socket) ->
    receive
       {udp, Socket, Host, Port, Bin} ->
           BinReply = ... ,
           gen_udp:send(Socket, Host, Port, BinReply),
           loop(Socket)
    end.
```

这比TCP程序要容易一些，因为无需担心如何让进程接收“套接字关闭”的消息。请注意，我们用binary模式打开了套接字，它告诉驱动要把所有消息以二进制数据的形式发送给控制进程。

接下来是客户端。这里有一个非常简单的客户端。它只是打开一个UDP套接字，向服务器发送一个消息，等待回复（或者超时），然后关闭套接字并返回服务器的返回值。

```
client(Request) ->
    {ok, Socket} = gen_udp:open(0, [binary]),
    ok = gen_udp:send(Socket, "localhost", 4000, Request),
    Value = receive
                {udp, Socket, _, _, Bin} ->
                    {ok, Bin}
            after 2000 ->
                error
            end,
    gen_udp:close(Socket),
    Value
```

必须设置一个超时，因为UDP是不可靠的，我们可能会得不到回复。

17.4.2　一个UDP阶乘服务器

可以很轻松地创建一个服务器，让它用发送给它的任意数字计算我们的老朋友——阶乘。这段代码建立在上一节的基础之上。

udp_test.erl

```
-module(udp_test).
-export([start_server/0, client/1]).

start_server() ->
    spawn(fun() -> server(4000) end).

%% 服务器
server(Port) ->
    {ok, Socket} = gen_udp:open(Port, [binary]),
    io:format("server opened socket:~p~n",[Socket]),
    loop(Socket).

loop(Socket) ->
    receive
        {udp, Socket, Host, Port, Bin} = Msg ->
```

```
            io:format("server received:~p~n",[Msg]),
            N = binary_to_term(Bin),
            Fac = fac(N),
            gen_udp:send(Socket, Host, Port, term_to_binary(Fac)),
            loop(Socket)
    end.

fac(0) -> 1;
fac(N) -> N * fac(N-1).

%% 客户端

client(N) ->
    {ok, Socket} = gen_udp:open(0, [binary]),
    io:format("client opened socket=~p~n",[Socket]),
    ok = gen_udp:send(Socket, "localhost", 4000,
                      term_to_binary(N)),
    Value = receive
                {udp, Socket, _, _, Bin} = Msg ->
                    io:format("client received:~p~n",[Msg]),
                    binary_to_term(Bin)
            after 2000 ->
                    0
            end,
    gen_udp:close(Socket),
    Value.
```

请注意，我添加了一些打印语句，这样就能看到程序运行时发生了什么。我在开发程序时总会添加一些打印语句，然后在程序能正常工作时编辑或注释掉它们。

现在来运行这个示例。首先，启动服务器。

```
1> udp_test:start_server().
server opened socket:#Port<0.106>
<0.34.0>
```

它在后台运行，这样我们就可以生成一个请求40阶乘值的客户端请求。

```
2> udp_test:client(40).
client opened socket=#Port<0.105>
server received:{udp,#Port<0.106>,{127,0,0,1},32785,<<131,97,40>>}
client received:{udp,#Port<0.105>,
                 {127,0,0,1}, 4000,
                 <<131,110,20,0,0,0,0,0,64,37,5,255,
                   100,222,15,8,126,242,199,132,27,
                   232,234,142>>}
815915283247897734345611269596115894272000000000
```

现在就得到了一个小小的UDP阶乘服务器。为了找点乐子，你可以试着编写这个程序的TCP版本，然后对它们进行基准测试来加以比较。

17.4.3 UDP数据包须知

要注意的是，因为UDP是一种无连接协议，所以服务器无法通过拒绝读取来自某个客户端的数据来阻挡它。服务器对谁是客户端一无所知。

大型UDP数据包可能会分段通过网络。当UDP数据经过网络上的路由器时，如果数据大小超过了路由器允许的最大传输单元（Maximum Transfer Unit，简称MTU）大小，分段就会发生。通常的建议是在调整UDP网络时从一个较小的数据包大小开始（比如大约500字节），然后逐步增大并测量吞吐量。如果吞吐量在某个时刻骤减，你就知道数据包太大了。

UDP数据包可以传输两次（这出乎一些人的意料之外），所以在编写远程过程调用代码时一定要小心。第二次查询得到的回复可能只是第一次查询回复的复制。为防止这类问题，可以修改客户端代码来加入一个唯一的引用，然后检查服务器是否返回了这个引用。要生成一个唯一的引用，需要调用Erlang的内置函数`make_ref`，它能确保返回一个全局唯一的引用。远程过程调用的代码现在看起来就像这样：

```
client(Request) ->
    {ok, Socket} = gen_udp:open(0, [binary]),
    Ref = make_ref(), %% 生成一个唯一的引用
    B1 = term_to_binary({Ref, Request}),
    ok = gen_udp:send(Socket, "localhost", 4000, B1),
    wait_for_ref(Socket, Ref).

wait_for_ref(Socket, Ref) ->
    receive
        {udp, Socket, _, _, Bin} ->
            case binary_to_term(Bin) of
                {Ref, Val} ->
                     %% 得到的是正确值
                     Val;
                {_SomeOtherRef, _} ->
                     %% 其他值则丢弃
                     wait_for_ref(Socket, Ref)
            end;
    after 1000 ->
          ...
    end.
```

以上就是UDP的相关介绍。UDP经常用于有低延迟要求的在线游戏，对它们来说是否偶尔丢包则无关紧要。

17.5 对多台机器广播

最后来看如何设立一个广播信道。它的代码很简单。

broadcast.erl

```
-module(broadcast).
-compile(export_all).

send(IoList) ->
    case inet:ifget("eth0", [broadaddr]) of
        {ok, [{broadaddr, Ip}]} ->
            {ok, S} =  gen_udp:open(5010, [{broadcast, true}]),
            gen_udp:send(S, Ip, 6000, IoList),
            gen_udp:close(S);
        _ ->
            io:format("Bad interface name, or\n"
                      "broadcasting not supported\n")
    end.

listen() ->
    {ok, _} = gen_udp:open(6000),
    loop().
loop() ->
    receive
        Any ->
            io:format("received:~p~n", [Any]),
            loop()
    end.
```

在这里需要两个端口，一个发送广播，另一个监听回应。我们选择了5010端口来发送广播请求，6000端口用来监听广播（这两个数字没有特殊含义，我只是选择了系统里两个空闲的端口）。

只有发送广播的进程才会打开5010端口，而网络上的所有机器都会调用`broadcast:listen()`来打开6000端口并监听广播消息。

`broadcast:send(IoList)`会对局域网里的所有机器广播`IoList`。

注意　要让它正常工作，接口名必须正确，而且系统必须支持广播。比如，我在iMac上使用了“en0”而非“eth0”。另外要注意的是，如果运行UDP监听程序的主机属于不同的子网，就不太可能收到UDP广播，因为路由器会默认丢弃这样的UDP广播。

17.6　一个 SHOUTcast 服务器

在这一章的最后，我们将运用新学到的套接字编程技术来编写一个SHOUTcast服务器。SHOUTcast是由Nullsoft公司开发的协议，它被用于传输音频数据流[①]。SHOUTcast使用HTTP作为传输协议来发送MP3或AAC编码的音频数据。

为了了解这一切是如何工作的，首先来看SHOUTcast协议，然后是服务器的整体架构，并在最后展示代码。

① http://www.shoutcast.com/

17.6.1 SHOUTcast协议

SHOUTcast协议很简单。

(1) 首先，客户端（例如XMMS、Winamp或iTunes）发送一个HTTP请求到SHOUTcast服务器。这是我在家里运行SHOUTcast服务器时XMMS生成的请求：

```
GET / HTTP/1.1
Host: localhost
User-Agent: xmms/1.2.10
Icy-MetaData:1
```

(2) SHOUTcast服务器的回复是：

```
ICY 200 OK
icy-notice1: <BR>This stream requires
  <a href=http://www.winamp.com/>;Winamp</a><BR>
icy-notice2: Erlang Shoutcast server<BR>
icy-name: Erlang mix
icy-genre: Pop Top 40 Dance Rock
icy-url: http://localhost:3000
content-type: audio/mpeg
icy-pub: 1
icy-metaint: 24576
icy-br: 96
... 数据 ...
```

(3) 现在SHOUTcast服务器会发送连续的数据流。这个数据流具有如下结构：

```
F H F H F H F ...
```

F是一个MP3音频数据块，它的长度必须刚好是24 576字节（icy-metaint参数的值）。H是一个数据头，由单字节的K后接16*K字节的数据组成。因此，可以用二进制型表示的最小数据头是<<0>>。接下来的数据头可以这样表示：

```
<<1,B1,B2, ..., B16>>
```

它的数据部分是一个StreamTitle=' ... ';StreamUrl='http:// ...';形式的字符串，长度不足则在右边补零，直到填满整个数据头。

17.6.2 SHOUTcast服务器的工作原理

要制作一个服务器，必须考虑到以下细节。

(1) 制作一个*播放列表*。我们在16.2.5节的“读取MP3元数据”部分创建了一个包含歌名清单的文件，服务器将使用这个文件。音频文件将从清单里随机选择。

(2) 制作一个并行服务器，这样就能并行传输多个流。我们将使用17.1.3节中“并行服务器”部分描述的方法。

(3) 我们只想发送各个音频文件的音频数据到客户端，而不包括内嵌的ID3标签。音频编码器理应能跳过损坏的数据，所以原则上可以把ID3标签和数据一起发送。然而在实践中，似乎需要移除ID3标签才能让程序更好地运行。

使用code/id3_tag_lengths.erl里的代码来移除标签，它位于本书可供下载的源代码中[①]。

17.6.3　SHOUTcast服务器的伪代码

在展示最终程序之前，先来看看省略了细节部分的整体代码流。

```
start_parallel_server(Port) ->
   {ok, Listen} = gen_tcp:listen(Port, ..),
   %% 创建一个歌曲服务器，它了解我们的所有音乐
   PidSongServer = spawn(fun() -> songs() end),
   spawn(fun() -> par_connect(Listen, PidSongServer) end).

%% 为每个连接分裂一个这样的进程
par_connect(Listen, PidSongServer) ->
    {ok, Socket} = gen_tcp:accept(Listen),
    %% 在accept返回时分裂一个新进程来等待下一个连接
    spawn(fun() -> par_connect(Listen, PidSongServer) end),
    inet:setopts(Socket, [{packet,0},binary, {nodelay,true},
                          {active, true}]),
    %% 处理请求
    get_request(Socket, PidSongServer, []).

%% 等待TCP请求
get_request(Socket, PidSongServer, L) ->
     receive
        {tcp, Socket, Bin} ->
           ... Bin包含来自客户端的请求
           ... 如果请求是分段的就再次调用循环，
           ... 否则调用got_request(Data, Socket,PidSongServer)
        {tcp_closed, Socket} ->
           ... 这是针对客户端在发送请求之前就已中止的情况（非常罕见）
 end.

%% 我们接到了请求，发送一个回复
 got_request(Data, Socket, PidSongServer) ->
      ... Data是来自客户端的请求...
      ... 分析它...
      ... 我们将始终满足请求...
      gen_tcp:send(Socket, [response()]),
      play_songs(Socket, PidSongServer).

%% 持续播放歌曲直到客户端退出
```

① http://pragprog.com/titles/jaerlang2/source_code

```
play_songs(Socket, PidSongServer) ->
    ... PidSongServer持有一份MP3文件清单
    Song = rpc(PidSongServer, random_song),
    ... Song是一首随机歌曲...
    Header = make_header(Song),
    ... 生成数据头...
    {ok, S} = file:open(File, [read,binary,raw]),
    send_file(1, S, Header, 1, Socket),
    file:close(S),
    play_songs(Socket, PidSongServer).
send_file(K, S, Header, OffSet, Socket) ->
  ... 把文件分块发送给客户端...
  ... 发送完整个文件就返回...
  ... 但如果写入套接字出错则退出,
  ... 这会发生在客户端退出时
```

如果你查看真实的代码，就会发现细节稍有不同，但它们的原理是相同的。完整的代码清单不会在这里展示，但可以在文件`code/shout.erl`里找到。

17.6.4　运行SHOUTcast服务器

要运行服务器并看看它能否工作，需要执行下列步骤。

(1) 制作一个播放列表。

(2) 启动服务器。

(3) 将一个客户端指向服务器。

制作列表需要以下三步。

(1) 移至代码目录。

(2) 编辑`mp3_manager.erl`文件里`start1`函数的路径，让它指向待输出音频文件所属目录的根目录。

(3) 编译`mp3_manager`，然后输入命令`mp3_manager:start1()`。应该能看到如下输出：

```
1> c(mp3_manager).
{ok,mp3_manager}
2> mp3_manager:start1().
Dumping term to mp3data
ok
```

如果有兴趣，现在可以查看`mp3data`文件来了解分析结果。

现在可以启动SHOUTcast服务器了。

```
1> shout:start().
...
```

要测试这个服务器，请执行下列操作。

(1) 去另一个窗口打开某个音频播放器，将它指向`http://localhost:3000`上的流服务。

我的系统使用XMMS，命令如下：

```
xmms http://localhost:3000
```

注意 如果想从另一台计算机访问这个服务器，就必须提供服务器所运行机器的IP地址。举个例子，要用我Windows机器上的Winamp访问服务器，我会打开Winamp的“播放”>“URL”菜单，然后在“打开URL”对话框里输入地址`http://192.168.1.168:3000`。

如果用的是iMac上的iTunes，我就会在“高级”>“打开流”菜单里输入上面的URL来访问服务器。

(2) 你会在启动服务器的窗口里看到一些诊断输出。

(3) 尽情享受吧！

在这一章，我们只介绍了最常用的套接字操作函数。可以在`gen_tcp`、`gen_udp`和`inet`的手册页里找到更多有关套接字API的信息。

有了一个简单的套接字接口，再加上内置函数`term_to_binary/1`和它的逆函数`binary_to_term`，网络编程就变得很容易了。建议你完成下面这些练习来获得第一手的体验。

在下一章，我们将来学习WebSocket。Erlang进程可以通过WebSocket直接与Web浏览器通信（无需遵循HTTP协议）。它是实现低延迟Web应用程序的理想方式，也提供了一种编写Web应用程序的简单方法。

17.7 练习

(1) 修改`nano_get_url/0`的代码（17.1.1节），并在必要时添加合适的HTTP头或执行重定向来获取任意网页。在多个网站上测试它。

(2) 输入17.1.2节里的代码，然后修改此代码来接收一个`{Mod, Func, Args}`元组（而不是字符串），最后计算`Reply = apply(Mod, Func, Args)`并把值发回套接字。

编写一个`nano_client_eval(Mod, Func, Args)`函数（类似于本章前面所展示的版本），让它用修改版服务器代码能理解的形式编码`Mod`、`Func`和`Arity`。

测试客户端和服务器代码能否正常工作，首先在同一台机器上，然后在同一局域网的两台机器上，最后在互联网上的两台机器上。

(3) 用UDP代替TCP重复上一个练习。

(4) 添加一个加密层，做法是先编码二进制型再发送给输出套接字，并在输入套接字接收之后立即解码。

(5) 制作一个简单的“类电子邮件”系统。把Erlang数据类型作为消息存储在`${HOME}/mbox`目录里。

第 18 章 用WebSocket和Erlang进行浏览

在这一章里，我们将来看看如何在浏览器里构建应用程序，把消息传递这个概念延伸到Erlang之外。通过这种方式，就可以轻松构建分布式应用程序，并将它们与Web浏览器整合在一起。Erlang认为Web浏览器只不过是另一个Erlang进程，这就简化了我们的编程模型，把一切都放进了同一个概念框架内。

假设Web浏览器是一个Erlang进程。如果我们想让浏览器做点什么，就会向它发送一个消息。如果浏览器里发生了需要我们处理的事情，它就会向我们发送一个消息。让这一切成为可能的是WebSocket（Web套接字）。WebSocket是HTML5标准的一部分，它是一种双向异步套接字，可以用来在浏览器和外部程序之间传递消息。对我们来说，这个外部程序就是Erlang运行时系统。

为了给Erlang运行时系统建立WebSocket接口，就会运行一个名为cowboy（牛仔）的简单Erlang服务器，让它管理套接字和WebSocket协议。第25章会详细介绍如何安装cowboy。为了简化讨论，我们假定Erlang和浏览器之间传递的所有消息都是JSON格式的。

这些消息在应用程序的Erlang端体现为Erlang映射组（参见5.3节），在浏览器里则体现为JavaScript对象。

在本章后续部分，我们将看到六个示例程序，它们包括在浏览器里运行的代码和在服务器里运行的代码。最后，来看客户端–服务器协议，了解它如何处理从Erlang到浏览器的消息。

运行这些示例需要三样东西：一些运行在浏览器里的代码，一些运行在Erlang服务器里的代码，以及一个能理解WebSocket协议的Erlang服务器。我们不会展示所有的代码，只针对在浏览器和服务器里运行的代码，服务器自身的代码也不会展示。所有示例都可以在https://github.com/joearms/ezwebframe里找到。示例里的浏览器代码只在Chrome浏览器里测试过。

注意 这里展示的代码是ezwebframe数据仓库所存代码的简化版，它们是用映射组编写的。数据仓库里的代码与Erlang分发套装保持同步，会在Erlang的R17版引入映射组后反映出Erlang的变化。

要亲自运行这些代码，你需要下载它们并执行安装交互操作。对我们而言，代码中有趣的部

分是运行在浏览器和服务器里的那些。

所有示例都使用一种简单的方法来让Erlang控制浏览器。如果Erlang想让浏览器做些什么，就向浏览器发送一个消息来告诉它。如果用户想要做些什么，就点击浏览器里的某个按钮或其他控件，生成一个消息发给Erlang。第一个示例详细展示了它的工作原理。

18.1 创建一个数字时钟

下图展示了一个运行在浏览器里的时钟。所有无关的浏览器窗口细节，比如菜单、工具栏和滚动条都没有显示出来，这样我们就能把注意力集中在代码上。



这个应用程序的关键部分是显示界面，它里面的时间会每秒钟更新一次。从Erlang的角度看，整个浏览器就是一个进程。因此，为了把时钟更新为之前显示的值，Erlang向浏览器发送了如下消息：

```
Browser ! #{ cmd => fill_div, id => clock, txt => <<"16:30:52">> }
```

我们在浏览器里载入了一张HTML网页，里面有一小段HTML：

```
<div id='clock'>
  ...
</div>
```

浏览器收到`fill_div`后就把它转换成JavaScript命令`fill_div({cmd:'fill_div', id:'clock', txt:'16:30:52'})`，后者随即把指派的字符串作为内容填充到div中。

请注意包含一个结构的Erlang消息是如何被转换成等价的JavaScript函数调用，然后在浏览器里执行的。扩展这个系统极其简单。你要做的就是编写一个JavaScript小函数来对应你需要处理的Erlang消息。

要完成这张图，需要添加启动和停止时钟的代码。把所有东西都放到一起后，HTML代码看起来就像这样：

websockets/clock1.html

```
<script type="text/javascript" src="./jquery-1.7.1.min.js"></script>
<script type="text/javascript" src="./websock.js"></script>
<link rel="stylesheet" href="./clock1.css" type="text/css">
<body>
  <div id="clock"></div>
```

```
  <button id="start" class="live_button">start</button>
  <button id="stop"  class="live_button">stop</button>
</body>
<script>
$(document).ready(function(){
   connect("localhost", 2233, "clock1");
   });
</script>
```

首先，载入了两个JavaScript库和一个样式表。clock1.css的作用是给时钟外观添加样式。接下来是一些构造外观的HTML。最后，我们有一小段JavaScript，它会在网页载入后运行。

注意　我们在所有示例里都假定你了解一些jQuery知识。jQuery（http://jquery.com）是一个极其流行的JavaScript库，它简化了在浏览器里操作对象的工作。

---

websock.js包含了所有必需的代码来打开WebSocket和连接浏览器DOM对象到Erlang。它会做下列事情。

(1) 给网页里所有属于live_button类的按钮添加点击处理函数。这些点击处理函数会在按钮被点击时向Erlang发送消息。

(2) 尝试启动一个到http://localhost:2233的WebSocket连接。在服务器端会有一个新分裂出的进程调用clock1:start(Browser)函数。所有这些都是通过调用JavaScript函数connect("localhost", 2233, "clock1")实现的。2233这个数字没有什么特别的含义，任何大于1023的未使用端口号都可以用。

现在是Erlang代码：

websockets/clock1.erl

```
-module(clock1).
-export([start/1, current_time/0]).

start(Browser) ->
    Browser ! #{ cmd => fill_div, id => clock, txt => current_time() },
    running(Browser).

running(Browser) ->
    receive
        {Browser, #{ clicked => <<"stop">>} } ->
            idle(Browser)
    after 1000 ->
            Browser ! #{ cmd => fill_div, id => clock, txt => current_time() },
            running(Browser)
    end.

idle(Browser) ->
    receive
        {Browser, #{clicked => <<"start">>} } ->
```

```
            running(Browser)
    end.

current_time() ->
    {Hour,Min,Sec} = time(),
    list_to_binary(io_lib:format("~2.2.0w:~2.2.0w:~2.2.0w",
                                 [Hour,Min,Sec])).
```

Erlang代码从`start(Browser)`开始执行。`Browser`是一个代表浏览器的进程。这段代码的第一个有趣之处如下：

```
Browser ! #{ cmd => fill_div, id => clock, txt => current_time() }
```

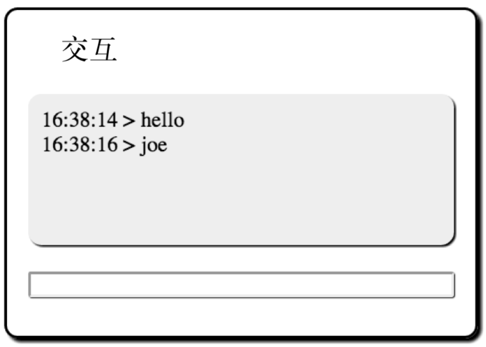

它会更新时钟的显示，我再次列出这一行是为了着重强调。编辑让我去掉它，没门。在我看来这是非常漂亮的代码。若希望让浏览器做点什么，就向它发送一个消息。就像在Erlang里一样。我们驯服了浏览器，它看起来就像是一个Erlang进程。太棒了！

初始化之后，`clock1`会调用`running/1`。如果收到一个`{clicked => <<"stop">>}`消息，就会调用`idle(Browser)`。否则，会在一秒钟的超时到期后向浏览器发送一个更新时钟的命令，然后调用自身。

`idle/1`等待一个`start`消息，然后调用`running/1`。

18.2　基本交互

接下来的示例有一个显示数据的可滚动文本区域和一个输入框。当你在输入框里输入文本并按回车时就会发送一个消息给浏览器。浏览器以一个更新显示内容的消息作为响应。



它的HTML代码如下：

websockets/interact1.html

```
<script type="text/javascript" src="./jquery-1.7.1.min.js"></script>
<script type="text/javascript" src="./websock.js"></script>
<link rel="stylesheet" href="./interact1.css" type="text/css">
<body>
```

```
  <h2>Interaction</h2>
  <div id="scroll"></div>
  <br>
  <input id="input" class="live_input"></input>
</body>
<script>
$(document).ready(function(){
   connect("localhost", 2233, "interact1");
   });
</script>
```

然后是Erlang代码：

websockets/interact1.erl

```
-module(interact1).
-export([start/1]).

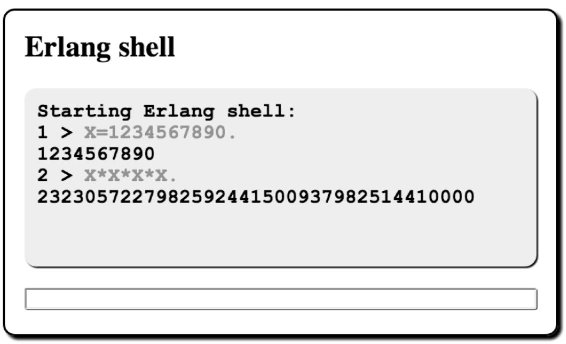
start(Browser) -> running(Browser).

running(Browser) ->
    receive
        {Browser, #{entry => <<"input">>, txt => Bin} }
            Time = clock1:current_time(),
            Browser ! #{cmd => append_div, id => scroll,
                        txt => list_to_binary([Time, " > ", Bin, "<br>"])}
    end,
    running(Browser).
```

它的工作方式类似于时钟示例。每当用户在输入框里按下回车键时，输入框就会发送一个包含输入文本的消息给浏览器。管理窗口的Erlang进程接收这个消息，然后向浏览器发回一个更新显示内容的消息。

18.3　浏览器里的 Erlang shell

可以用接口模式里的代码制作一个在浏览器里运行的Erlang shell。



我们不会展示全部代码，因为它和交互示例的代码差不多。下面是一些相关部分的代码：

websockets/shell1.erl

```
start(Browser) ->
    Browser ! #{cmd => append_div, id => scroll,
                txt => <<"Starting Erlang shell:<br>">>},
    B0 = erl_eval:new_bindings(),
    running(Browser, B0, 1).
running(Browser, B0, N) ->
receive
    {Browser, #{entry => <<"input">>}, txt => Bin}} ->
        {Value, B1} = string2value(binary_to_list(Bin), B0),
        BV = bf("~w > <font color='red'>~s</font><br>~p<br>",
                [N, Bin, Value]),
        Browser ! #{cmd => append_div, id => scroll, txt => BV},
        running(Browser, B1, N+1)
end.
```

难点由代码里解析输入字符串并求值的函数完成。

websockets/shell1.erl

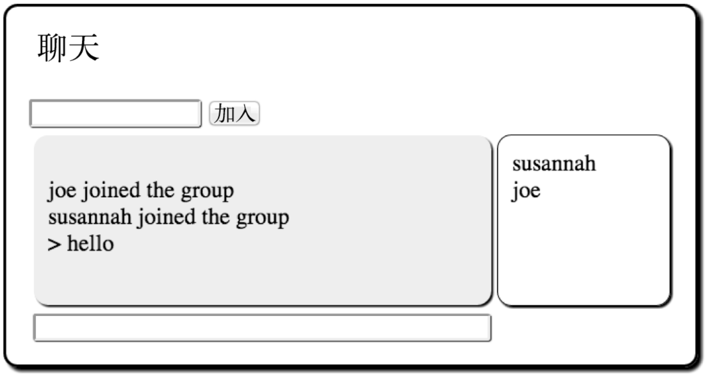
```
string2value(Str, Bindings0) ->
    case erl_scan:string(Str, 0) of
      {ok, Tokens, _} ->
        case erl_parse:parse_exprs(Tokens) of
          {ok, Exprs} ->
            {value, Val, Bindings1} = erl_eval:exprs(Exprs, Bindings0),
              {Val, Bindings1};
            Other ->
              io:format("cannot parse:~p Reason=~p~n",[Tokens,Other]),
                        {parse_error, Bindings0}
        end;
      Other ->
        io:format("cannot tokenise:~p Reason=~p~n",[Str,Other])
    end.
```

现在就有了一个在浏览器里运行的Erlang shell。诚然，这是一个非常初步的shell，但它演示了构建更复杂shell所需的全部技巧。

18.4 创建一个聊天小部件

在本章接下来的部分里，我们将开发一个IRC[①]控制程序。这个程序需要一个聊天小部件。

① IRC是Internet Relay Chat（互联网中继聊天）的缩写，它以客户端–服务器的形式进行文本消息传输。——译者注



创建这个小部件的代码如下：

websockets/chat1.html

```
<script type="text/javascript" src="./jquery-1.7.1.min.js"></script>
<script type="text/javascript" src="./websock.js"></script>
<link rel="stylesheet" href="./chat1.css" type="text/css">

<body>
  <h2>Chat</h2>
  <input id="nick_input"/>
  <button id="join">Join</button>
  <br/>
  <table>
    <tr>
      <td><div id="scroll"></div></td>
      <td><div id="users"></div></td>
    </tr>
    <tr>
      <td colspan="2">
        <input id="tell" class="live_input"/>
      </td>
    </tr>
  </table>
</body>

<script>
$(document).ready(function(){
  $("#join").click(function(){
      var val = $("#nick_input").val();
      send_json({'join':val});
      $("#nick_input").val("");
      });
   connect("localhost", 2233, "chat1");
});
</script>
```

这段代码和前面几个例子中的大致相同，唯一的区别是"Join"按钮的用法。我们想让"Join"按钮在点击后执行一个浏览器本地操作，而不是向控制程序发送一个消息。下面的代码用jQuery做到了这一点：

```
$("#join").click(function(){
    var val = $("#nick_input").val();
    send_json({'join':val});
    $("#nick_input").val("");
  })
```

这段代码给"Join"按钮挂接了一个事件处理器。点击"Join"按钮，读取昵称输入字段，并向Erlang发送一个`join`消息，然后清除输入框。

Erlang端要做的事同样很简单。我们必须响应两类消息：一类是用户点击"Join"按钮时发送的`join`消息；另一类是用户在小部件底部的输入字段里按下回车后发送的`tell`消息。

要测试这个小部件，可以使用如下代码：

websockets/chat1.erl

```
-module(chat1).
-export([start/1]).

start(Browser) ->
    running(Browser, []).

running(Browser, L) ->
    receive
        {Browser, #{join => Who}} ->
            Browser ! #{cmd => append_div ,id => scroll,
                        txt => list_to_binary([Who, " joined the group\n"])},
            L1 = [Who,"<br>"|L],
            Browser ! #{cmd => fill_div, id => users,
                        txt => list_to_binary(L1)},
            running(Browser, L1);
        {Browser,#{entry => <<"tell">>, txt => Txt}} ->
            Browser ! #{cmd => append_div, id => scroll,
                        txt => list_to_binary([" > ", Txt, "<br>"])},
            running(Browser, L);
        X ->
            io:format("chat received:~p~n",[X])
    end,
    running(Browser, L).
```

这不是控制IRC应用程序的真实代码，而是一个测试程序。当它收到`join`消息时，滚动区域就会更新，用户div里的用户名单也会发生变化。当它收到`tell`消息时，只有滚动区域会发生变化。

18.5　简化版 IRC

上一节里的聊天小部件可以轻松扩展成一个更真实的聊天程序。为做到这一点，我们将把聊天小部件的代码修改如下：

websockets/chat2.html

```
<script type="text/javascript" src="./jquery-1.7.1.min.js"></script>
<script type="text/javascript" src="./websock.js"></script>
<link rel="stylesheet" href="./chat1.css" type="text/css">

<body>
  <h2>Chat</h2>
  <div id="idle">
    <input id="nick_input"/>
    <button id="join">Join</button>
    <br/>
  </div>

  <div id="running">
    <table>
      <tr>
        <td><div id="scroll"></div></td>
        <td><div id="users"></div></td>
      </tr>
      <tr>
        <td colspan="2">
          <input id="tell" class="live_input"/><br/>
          <button class="live_button" id="leave">Leave</button>
        </td>
      </tr>
    </table>
  </div>
</body>

<script>
$(document).ready(function(){
  $("#running").hide();
  $("#join").click(function(){
      var val = $("#nick_input").val();
      send_json({'join':val});
      $("#nick_input").val("");
      });

  connect("localhost", 2233, "chat2");
  });

function hide_div(o){
  $("#" + o.id).hide();
```

```
}

function show_div(o){
  $("#" + o.id).show();
}
</script>
```

这段代码有两个主要的div，分别是`idle`（空闲）和`running`（运行）。它们中有一个隐藏，另一个显示。当用户点击“Join”按钮时会生成一个发送到IRC服务器的请求来加入聊天。如果用户名未被使用，服务器就会回复一个欢迎消息，然后聊天处理程序会隐藏`idle`div并显示`running`div。对应的Erlang代码如下：

websockets/chat2.erl

```
-module(chat2).
-export([start/1]).

start(Browser) ->
    idle(Browser).

idle(Browser) ->
    receive
        {Browser, #{join => Who}} ->
            irc ! {join, self(), Who},
            idle(Browser);
        {irc, welcome, Who} ->
            Browser ! #{cmd => hide_div, id => idle},
            Browser ! #{cmd => show_div, id => running},
            running(Browser, Who);
        X ->
            io:format("chat idle received:~p~n",[X]),
            idle(Browser)
    end.

running(Browser, Who) ->
    receive
        {Browser,#{entry => <<"tell">>, txt => Txt}} ->
            irc ! {broadcast, Who, Txt},
            running(Browser, Who);
        {Browser,#{clicked => <<"Leave">>}} ->
            irc ! {leave, Who},
            Browser ! #{cmd => hide_div, id => running},
            Browser ! #{cmd => show_div, id => idle},
            idle(Browser);
        {irc, scroll, Bin} ->
            Browser ! #{cmd => append_div, id => scroll, txt => Bin},
            running(Browser, Who);
```

```
        {irc, groups, Bin} ->
            Browser ! #{cmd => fill_div, id => users, txt => Bin},
            running(Browser, Who);
        X ->
            io:format("chat running received:~p~n",[X]),
            running(Browser, Who)
    end.
```

在运行状态时，聊天控制程序可以接收四类消息。其中两类来自浏览器：说话（tell）消息是在用户把消息输入到聊天输入字段之后收到的；离开（leave）消息是在用户点击“Leave”按钮之后收到的。这些消息会被中继到IRC服务器上，如果是离开消息，就会有后续的消息发送给浏览器来隐藏`running` div并显示`idle` div，这样就回到初始状态了。

其余两类消息来自IRC服务器，它们会让控制程序更新滚动区域或者用户名单。

IRC控制程序的代码非常简单。

websockets/irc.erl

```
-module(irc).
-export([start/0]).

start() ->
    register(irc, spawn(fun() -> start1() end)).

start1() ->
    process_flag(trap_exit, true),
    loop([]).

loop(L) ->
    receive
        {join, Pid, Who} ->
            case lists:keysearch(Who,1,L) of
                false ->
                    L1 = L ++ [{Who,Pid}],
                    Pid ! {irc, welcome, Who},
                    Msg = [Who, <<" joined the chat<br>">>],
                    broadcast(L1, scroll, list_to_binary(Msg)),
                    broadcast(L1, groups, list_users(L1)),
                    loop(L1);
                {value,_} ->
                    Pid ! {irc, error, <<"Name taken">>},
                    loop(L)
            end;
        {leave, Who} ->
            case lists:keysearch(Who,1,L) of
                false ->
                    loop(L);
                {value,{Who,Pid}} ->
                    L1 = L -- [{Who,Pid}],
                    Msg = [Who, <<" left the chat<br>">>],
```

```
                    broadcast(L1, scroll, list_to_binary(Msg)),
                    broadcast(L1, groups, list_users(L1)),
                    loop(L1)
            end;
        {broadcast, Who, Txt} ->
            broadcast(L, scroll,
                      list_to_binary([" > ", Who, " >> ", Txt, "<br>"])),
            loop(L);
        X ->
            io:format("irc:received:~p~n",[X]),
            loop(L)
    end.
broadcast(L, Tag, B) ->
    [Pid ! {irc, Tag, B} || {_,Pid} <- L].

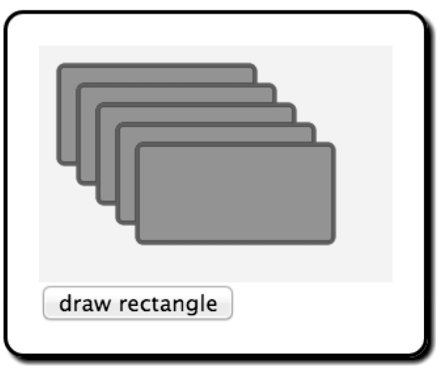
list_users(L) ->
    L1 = [[Who,"<br>"] || {Who,_}<- L],
    list_to_binary(L1).
```


这段代码需要解释一下。如果IRC服务器收到一个加入消息而用户名未被使用，它就会广播一个新的用户名单给所有已连接用户，同时广播一个加入消息给所有已连接用户的滚动区域。如果收到的是离开消息，它就会把该用户移出当前用户名单，并把这个信息广播给所有已连接用户。

要在一个分布式系统里运行它，就需要让一台机器运行IRC服务器。其他所有机器都必须知道服务器运行机器的IP地址或主机名。举个例子，如果IRC服务器运行机器的IP地址是`AAA.BBB.CCC.DDD`，那么其他所有机器都应该请求URL为`http://AAA.BBB.CCC.DDD:2233/chat2.html`（2233端口是默认的端口号）的网页。

18.6 浏览器里的图形

到目前为止，我们只在浏览器里见过文字。一旦把图形对象引入浏览器，一个全新的世界就出现了。要实现它其实很简单，因为可以使用内建于现代浏览器里的可伸缩矢量图形（Scalable Vector Graphics，简称SVG）格式。



以下是我们打算做的。在浏览器里创建一张SVG画布，并在下方放置一个按钮。点击按钮后会给Erlang发送一个消息。Erlang回复一个命令，让浏览器给SVG画布添加一个矩形。上面的屏幕截图展示了点击五次“draw rectangle”按钮后小部件的样子。实现它的Erlang代码如下：

websockets/svg1.erl

```
-module(svg1).
-export([start/1]).

start(Browser) ->
    Browser ! #{cmd => add_canvas, tag => svg, width => 180, height => 120},
    running(Browser, 10, 10).

running(Browser, X, Y) ->
    receive
        {Browser,#{clicked => <<"draw rectangle">>}} ->
            Browser ! #{cmd => add_svg_thing, type => rect,
                        rx => 3, ry => 3, x => X, y => Y,
                        width => 100,height => 50,
                        stroke => blue,'stroke-width' => 2,
                        fill =>  red},
            running(Browser, X+10, Y+10)
    end.
```

这段Erlang代码向浏览器发送两类消息：`[{cmd,add_canvas}, ...`和`[{cmd,add_svg_thing}, ...]`。因此，必须在浏览器载入的JavaScript里提供这些命令的定义。

websockets/svg1.html

```
<script type="text/javascript" src="./jquery-1.7.1.min.js"></script>
<script type="text/javascript" src="./websock.js"></script>
<link rel="stylesheet" href="./svg1.css" type="text/css">
<body>
  <div id="svg"></div>
  <button id="start" class="live_button">draw rectangle</button>
</body>

<script>
$(document).ready(function(){
   connect("localhost", 2233, "svg1");
   });

var canvas;
var svg_ns='http://www.w3.org/2000/svg';

function add_canvas(o){
    canvas = document.createElementNS(svg_ns, 'svg');
    canvas.setAttribute("width",  o.width);
    canvas.setAttribute("height", o.height );
    canvas.setAttribute("style",  "background-color:#eeffbb");
```

```
    $('#'+o.tag).append(canvas);
}

function add_svg_thing(o){
    var obj = document.createElementNS(svg_ns, o.type);
    for(key in o){
        var val = o[key];
        obj.setAttributeNS(null, key, val);
    };
    canvas.appendChild(obj);
}
</script>
```


有了这两个函数，就可以开始使用SVG图形了。请注意，当向浏览器发送一个命令来绘制矩形时，这个命令包含了许多参数。rx和ry等属性的作用是生成矩形的圆角。这些属性的完整清单可以在W3C规范[1]里找到。

18.7 浏览器-服务器协议

浏览器–服务器协议极其简单。它使用WebSocket来传输JSON消息，这种方式对Erlang和JavaScript都非常合适，让建立浏览器到Erlang的接口变得十分简单。

18.7.1 从 Erlang 发送消息到浏览器

要改变浏览器里的内容，可以从Erlang发送一个消息到浏览器。假设浏览器里有一个div，它的声明如下：

```
<div id="id123"></div>
```

要把这个div的内容改成字符串abc，Erlang可以向连接浏览器的WebSocket发送以下JSON消息：

```
[{cmd:'fill_div', id:'id123', txt:'abc'}]
```

当浏览器收到来自WebSocket的消息时，会激活webserver.js里名为onMessage的回调函数。调用它的代码如下：

```
websocket = new WebSocket(wsUri);
...
websocket.onmessage = onMessage;
```

这个回调函数的定义如下：

```
function onMessage(evt) {
    var json = JSON.parse(evt.data);
```

[1] http://www.w3.org/TR/SVG/

```
    do_cmds(json);
}

function do_cmds(objs){
  for(var i = 0; i < objs.length; i++){
     var o = objs[i];
     if(eval("typeof("+o.cmd+")") == "function"){
       eval(o.cmd + "(o)");
     } else {
       alert("bad_command:"+o.cmd);
     };
  };
}
```

`do_cmds(objs)`期望收到一列以下形式的命令：

```
[{cmd:command1, ...:..., ...:...},
 {cmd:command2, ...:..., ...:...},
 ...
 {cmd:commandN, ...:..., ...:...}]
```

列表里的每个命令都是一个JavaScript对象，它必须包含一个名为`cmd`的键。对列表里的每个对象`x`，系统都会检查是否存在名为`x.cmd`的函数。如果存在，它就会调用`x.cmd(x)`。因此，`{cmd:'fill_div', id:'id123', txt:'abc'}`会导致系统调用：

```
fill_div({cmd:'fill_div', id:'id123', txt:'abc'})
```

这种编码和执行命令的方式能够轻松扩展，这样就能在必要时给接口添加更多的命令。

18.7.2　从浏览器到 Erlang 的消息

当我们在浏览器里点击某个按钮时，就会执行这样的命令：

```
send_json({'clicked':txt}));
```

`send_json(x)`把参数`x`编码成JSON数据类型，然后把它写入WebSocket。这个消息在`websocket.erl`里接收并转换成一个结构，然后发送给管理WebSocket的控制进程。

我们已经了解了如何将消息传递的概念延伸到Erlang之外，以及如何利用消息传递来直接控制浏览器。从Erlang程序员的角度看，这个世界现在是一个秩序井然的空间，所有的对象都能响应Erlang消息。我们并不是在Erlang里做一套，在Erlang外做另一套。这为程序增加了很强的秩序感和统一性，使这个复杂的世界看起来更简单了。Web浏览器自然是一个相当复杂的对象，但是通过让它以可预测的方式响应少量消息，就能轻松控制并驾驭这种复杂性，从而构建强大的应用程序。

在这一章里，Web浏览器看起来就像一个Erlang进程。我们可以向浏览器发送消息来让它做许多事，同时当浏览器里发生了某些状况时，我们也能收到消息。在Erlang里用映射组代表消息，它们会被编码成JSON消息，然后在浏览器里体现为JavaScript对象。消息的表现形式在Erlang和

JavaScript里只有微小的概念差异，这就简化了编程工作，因为不必在改变表现形式的细节上过于费心。

现在我们将转向另一个主题。接下来的两章涉及存储大容量的数据。其中第19章详细介绍了底层存储模块`ets`和`dets`。第20章详细介绍了Erlang的`mnesia`数据库，它是用`ets`和`dets`实现的。

18.8 练习

(1) `shell1.erl`里启动的进程不够完善。如果它崩溃了，Web应用程序就会锁定并无法运行。给这个应用程序添加错误恢复代码。添加历史命令重放的功能。

(2) 阅读`websockets.js`里的代码，仔细追踪浏览器里某个活动按钮被点击后发生的事。跟着代码从JavaScript到WebSocket，再从WebSocket到Erlang。点击按钮后生成的消息是如何找到相应的Erlang控制进程的？

(3) 简化版IRC程序是一个功能完备的聊天程序。试着运行它并检查它的功能是否正常。你可能会发现防火墙之类的因素阻止它访问服务器，让它无法正常工作。如果是这样，就进行调查并看看能否开放防火墙。尝试找到真正的IRC协议规范，你会发现它比这里的版本长很多。为什么会这样？用某种用户身份验证系统来扩展这个IRC系统。

(4) IRC程序使用了一台中央服务器。能否修改这个程序，让它用点对点网络取代中央服务器？能否给聊天客户端添加SVG图形，或者用HTML5里的音频接口发送和接收声音数据？

第 19 章 用ETS和DETS存储数据

ets和dets是两个系统模块，可以用来高效存储海量的Erlang数据。ETS是Erlang Term Storage（Erlang数据存储）的缩写，DETS则是Disk ETS（磁盘ETS）的缩写。

ETS和DETS执行的任务基本相同：它们提供大型的键–值查询表。ETS常驻内存，DETS则常驻磁盘。ETS是相当高效的：可以用它存储海量的数据（只要有足够的内存），执行查找的时间也是恒定的（在某些情况下是对数时间）。DETS提供了几乎和ETS一样的接口，但它会把表保存在磁盘上。因为DETS使用磁盘存储，所以它远远慢于ETS，但是运行时的内存占用也会小很多。另外，ETS和DETS表可以被多个进程共享，这就让跨进程的公共数据访问变得非常高效。

ETS和DETS表是把键和值关联到一起的数据结构。最常用的表操作是*插入*和*查找*。ETS或DETS表其实就是Erlang元组的集合。

ETS表里的数据保存在内存里，它们是*易失的*。当ETS表被丢弃或者控制它的Erlang进程终止时，这些数据就会被删除。保存在DETS表里的数据是*非易失的*，即使整个系统崩溃也能留存下来。DETS表在打开时会进行一致性检查，如果发现有损坏，系统就会尝试修复它（这可能会花费很长时间，因为表里的所有数据都要检查）。

这应该能恢复表里的所有数据，但如果表里的最后一项是在系统崩溃时生成的，就可能会丢失。

ETS表广泛应用于那些必须以高效方式操作大量数据的应用程序，以及用非破坏性赋值和“纯”Erlang数据结构编程的开销过大之时。

ETS表看上去像是用Erlang实现的，但事实上它们是在底层的运行时系统里实现的，有着不同于普通Erlang对象的性能特点。特别需要指出的是，ETS表没有垃圾收集机制，这就意味着即使ETS表极其巨大也不会有垃圾收集的负担。不过，创建或访问ETS对象仍然会带来少许开销。

19.1 表的类型

ETS和DETS表保存的是元组。元组里的某一个元素（默认是第一个）被称为该表的键。通过键来向表里插入和提取元组。当我们向表里插入一个元组时会发生什么，取决于表的类型和键的值。一些表被称为*异键表*（set），它们要求表里所有的键都是唯一的。另一些被称为*同键表*（bag），它们允许多个元素拥有相同的键。

选择正确类型的表对应用程序的性能意义重大。

基本的表类型（异键表和同键表）各有两个变种，它们共同构成四种表类型：异键、有序异键（ordered set）、同键和副本同键（duplicate bag）。在异键表里，各个元组里的键都必须是独一无二的。在有序异键表里，元组会被排序。在同键表里可以有不止一个元组拥有相同的键，但是不能有两个完全相同的元组。在副本同键表里可以有多个元组拥有相同的键，而且在同一张表里可以存在多个相同的元组。

ETS和DETS表有四种基本操作。

- 创建一个新表或打开现有的表

  用`ets:new`或`dets:open_file`实现。
- 向表里插入一个或多个元组

  这里要调用`insert(TableId, X)`，其中X是一个元组或元组列表。`insert`在ETS和DETS里有着相同的参数和工作方式。
- 在表里查找某个元组

  这里要调用`lookup(TableID, Key)`。得到的结果是一个匹配Key的元组列表。`lookup`在ETS和DETS里都有定义。

  `lookup`的返回值始终是一个元组列表，这样就能对异键表和同键表使用同一个查找函数。如果表的类型是同键，那么多个元组可以拥有相同的键。如果表的类型是异键，那么查找成功后的列表里只会有一个元素。我们将在下一节介绍表的类型。

  如果表里没有任何元组拥有所需的键，就会返回一个空列表。
- 丢弃某个表

  用完某个表后可以告知系统，方法是调用`dets:close(TableId)`或`ets:delete(TableId)`。

可以用下面这个小测试程序来演示它们是如何工作的：

ets_test.erl

```
-module(ets_test).
-export([start/0]).

start() ->
    lists:foreach(fun test_ets/1,
                  [set, ordered_set, bag, duplicate_bag]).

test_ets(Mode) ->
    TableId = ets:new(test, [Mode]),
    ets:insert(TableId, {a,1}),
    ets:insert(TableId, {b,2}),
    ets:insert(TableId, {a,1}),
    ets:insert(TableId, {a,3}),
    List = ets:tab2list(TableId),
    io:format("~-13w  => ~p~n", [Mode, List]),
    ets:delete(TableId).
```

19

这个程序会为四种模式各创建一个ETS表，然后向表内插入{a,1}、{b,2}、{a,1}和{a,3}。然后调用tab2list，它会把整个表转换成一个列表并打印出来。

运行这个程序会得到以下输出：

```
1> ets_test:start().
set          => [{b,2},{a,3}]
ordered_set  => [{a,3},{b,2}]
bag          => [{b,2},{a,1},{a,3}]
duplicate_bag => [{b,2},{a,1},{a,1},{a,3}]
```

在异键型的表里，每个键都只能出现一次。如果先向表内插入元组{a,1}然后再插入{a,3}，那么最终值将是{a,3}。异键表和有序异键表的唯一区别是有序异键表里的元素会根据它们的键排序。用tab2list把表转换成列表之后就能看到这一顺序了。

同键型的表可以有多个相同的键。举个例子，当我们先插入{a,1}再插入{a,3}时，同键表就会同时包含这两个元组，而不仅仅是后一个。副本同键表里允许有多个完全相同的元组，这样当连续插入两次{a,1}时，表里就会有两个相同的{a,1}元组，而普通同键表里只会留下一个。

19.2　影响 ETS 表效率的因素

ETS表在内部是用散列表表示的（除了有序异键表，它是用平衡二叉树表示的）。这就意味着使用异键表会带来少许空间开销，而使用有序异键表会带来时间开销。插入异键表所需的时间是恒定的，而插入有序异键表所需的时间与表内条目数量的对数成比例。

当你在异键表和有序异键表之间进行选择时，应该考虑构建表之后会拿它做什么。如果想要一个排过序的表，就应该使用有序异键表。

使用同键表的代价比使用副本同键表更高，因为每次插入时都需要与所有拥有相同键的元素比较是否相等。如果有大量元组都拥有同一个键，这么做就会很低效。

ETS表保存在一个单独的存储区域里，与正常的进程内存无关。可以认为ETS表归创建它的进程所有，当这个进程挂了或者调用ets:delete时，表就会被删除。ETS表不会进行垃圾收集，这就意味着即使表里存储了海量的数据也不会产生垃圾收集的开销。

当一个元组被插入ETS表时，所有代表这个元组的数据结构都会从进程的栈复制到ETS表里。当你对表执行查询操作时，找到的元组会从ETS表复制到进程的栈上。

所有的数据结构都是如此，除了大型的二进制数据。这些二进制型会存储在它们自己的堆外存储区域里。这个区域可以被多个进程和ETS表共享，各个二进制型则由一个引用计数垃圾收集器管理，它会记录有多少个不同的进程和ETS表在使用这个二进制型。如果某个特定二进制型的进程和表使用计数降至零，那么这个二进制型的存储区域就可以被回收。

所有这些听上去可能比较复杂，但最终的结论是：在进程间传递包含大型二进制数据的消息是非常高效的，向ETS表插入包含二进制型的元组也非常高效。一个不错的经验法则是尽可能多用二进制型来表示字符串和大块的无类型内存。

19.3　创建一个 ETS 表

创建ETS表的方法是调用`ets:new`。创建表的进程被称为该表的主管。创建表时所设置的一组选项在以后是无法更改的。如果主管进程挂了，表的空间就会被自动释放。你可以调用`ets:delete`来删除表。

`ets:new`的参数如下。

```
-spec ets:new(Name, [Opt]) -> TableId
```

其中，`Name`是一个原子。`[Opt]`是一列选项，源于下面这些参数。

- `set | ordered_set | bag | duplicate_bag`
  创建一个指定类型的ETS表（我们之前讨论过）。
- `private`
  创建一个私有表，只有主管进程才能读取和写入它。
- `public`
  创建一个公共表，任何知道此表标识符的进程都能读取和写入它。
- `protected`
  创建一个受保护表，任何知道此表标识符的进程都能读取它，但只有主管进程才能写入它。
- `named_table`
  如果设置了此选项，`Name`就可以被用于后续的表操作。
- `{keypos, K}`
  用`K`作为键的位置。通常键的位置是1。基本上唯一需要使用这个选项的场合是保存Erlang记录（它其实是元组的另一种形式），并且记录的第一个元素包含记录名的时候。

注意　不使用任何选项打开一个ETS表就相当于用`[set,protected,{keypos,1}]`选项打开它。

本章里的所有代码都使用`protected`ETS表。受保护表特别有用，因为它们能实现几乎零成本的数据共享。所有知道此表标识符的本地进程都能读取数据，但只有一个进程能修改表里的数据。

类似黑板的ETS表

受保护表提供了一种“黑板系统”样式。你可以把受保护表想象成一种有名字的黑板。任何知道黑板名的人都可以阅读它，但只有所有者才能在黑板上写字。

注意：用`public`模式打开的ETS表可以被任何知道表名的进程读写。在这种情况下，用户必须确保以互不冲突的方式执行表的读取和写入操作。

19.4　ETS 示例程序

本节里的示例是关于生成三字母组合（trigram）的。这是一个不错的“表演性”程序，它演

示了ETS表的威力。

我们的目标是编写一个启发式程序，让它尝试预测某个给定字符串是否是英语单词。

为了预测某个随机字母序列是否是英语单词，我们将分析这个词里出现了哪些*三字母组合*。三字母组合是由三个连续字母组成的序列，但不是任何三字母序列都会在正确的英语单词里出现。举个例子，任何英语单词里都不会出现akj和rwb这样的三字母组合。因此，为了测试某个字符串是否是英语单词，要做的就是将字符串里所有连续的三个字母序列与一批三字母组合进行对比测试，后者是用一个大型英语单词集合生成的。

程序做的第一件事是用一个非常大的单词集合来计算英语中所有的三字母组合。要做到这一点，我们使用ETS异键表。这个决定是根据ETS异键表、有序异键表和`sets`模块所提供的"纯"Erlang异键表之间的一组相对性能测量结果得出的。

我们将在接下来的几节里做这些事。

(1) 制作一个遍历英语里所有三字母组合的迭代函数。这将大大简化向不同表类型插入三字母组合的编码工作。

(2) 创建`set`和`ordered_set`类型的ETS表来存放所有这些三字母组合。另外再创建一个异键表来存放这些三字母组合。

(3) 测量*创建*这几种表所需的时间。

(4) 测量*访问*这几种表所需的时间。

(5) 根据*测量结果*选择最佳的方法，并为它编写访问函数。

所有代码都在`lib_trigrams`里。我们将分几节呈现它，同时省略一些细节。别担心，完整的代码在本书主页[①]所提供的`code/lib_trigrams.erl`文件里。

19.4.1　三字母组合迭代函数

定义一个名为`for_each_trigram_in_the_english_language(F, A)`的函数。这个函数会把fun `F`应用到英语里的每一个三字母组合上。`F`是一个类型为`fun(Str, A) -> A`的fun，`Str`涵盖了英语里所有的三字母组合，`A`则是一个累加器。

要编写这个迭代函数，需要一个庞大的单词表。（注意：我在这里称它为迭代函数，但是更严格来说，它其实是一个很像`lists:foldl`的折叠操作符。）我使用了一个包含354 984个英语单词的集合[②]来生成三字母组合。利用这个单词表，可以定义三字母组合的迭代函数如下：

lib_trigrams.erl

```
for_each_trigram_in_the_english_language(F, A0) ->
    {ok, Bin0} = file:read_file("354984si.ngl.gz"),
    Bin = zlib:gunzip(Bin0),
    scan_word_list(binary_to_list(Bin), F, A0).
```

① http://pragprog.com/titles/jaerlang2/source_code

② http://www.dcs.shef.ac.uk/research/ilash/Moby/

```
scan_word_list([], _, A) ->
    A;
scan_word_list(L, F, A) ->
    {Word, L1} = get_next_word(L, []),
    A1 = scan_trigrams([$\s|Word], F, A),
    scan_word_list(L1, F, A1).

%% 扫描单词，寻找\r\n。
%% 第二个参数是（反转的）单词，
%% 所以必须在找到\r\n或扫描完字符时把它反转回来

get_next_word([$\r,$\n|T], L) -> {reverse([$\s|L]), T};
get_next_word([H|T], L)       -> get_next_word(T, [H|L]);
get_next_word([], L)          -> {reverse([$\s|L]), []}.

scan_trigrams([X,Y,Z], F, A) ->
    F([X,Y,Z], A);
scan_trigrams([X,Y,Z|T], F, A) ->
    A1 = F([X,Y,Z], A),
    scan_trigrams([Y,Z|T], F, A1);
scan_trigrams(_, _, A) ->
    A.
```

这里有两点要注意。首先，用`zlib:gunzip(Bin)`解压缩源文件里的二进制数据。这个单词表相当长，所以我们更希望让它以压缩文件的形式保存在磁盘上，而不是原始的ASCII文件。其次，在每个单词的前后各加了一个空格。进行三字母组合分析时，我们希望把空格当成一个普通字母。

19.4.2 创建一些表

我们将像这样创建ETS表：

lib_trigrams.erl

```
make_ets_ordered_set() -> make_a_set(ordered_set, "trigramsOS.tab").
make_ets_set()         -> make_a_set(set, "trigramsS.tab").

make_a_set(Type, FileName) ->
    Tab = ets:new(table, [Type]),
    F = fun(Str, _) -> ets:insert(Tab, {list_to_binary(Str)}) end,
    for_each_trigram_in_the_english_language(F, 0),
    ets:tab2file(Tab, FileName),
    Size = ets:info(Tab, size),
    ets:delete(Tab),
    Size.
```

请注意，当我们分离出一个三字母组合ABC时，实际上是把元组`{<<"ABC">>}`插入了代表各种三字母组合的ETS表里。这看起来很滑稽：元组里只有一个元素。元组通常是多个元素的容器，所以让元组只包含一个元素是违反常识的。但是请记住，ETS表里的所有条目都是元组，而且在

默认情况下元组的键就是它的第一个元素。所以对我们来说，{Key}代表了一个没有值的键。

接下来的代码将创建一个包含所有三字母组合的异键表（这次用的是Erlang模块sets，而不是ETS）：

lib_trigrams.erl

```
make_mod_set() ->
    D = sets:new(),
    F = fun(Str, Set) -> sets:add_element(list_to_binary(Str),Set) end,
    D1 = for_each_trigram_in_the_english_language(F, D),
    file:write_file("trigrams.set", [term_to_binary(D1)]).
```

19.4.3　创建表所需的时间

本章最后列出的lib_trigrams:make_tables()函数负责创建所有的表。它包含一些测试功能，这样就能测量表的大小以及创建表所需的时间。

```
1> lib_trigrams:make_tables().
Counting - No of trigrams=3357707 time/trigram=0.577938
Ets ordered Set size=19.0200 time/trigram=2.98026
Ets set size=19.0193 time/trigram=1.53711
Module Set size=9.43407 time/trigram=9.32234
ok
```

它告诉我们一共有330万个三字母组合，以及处理单词表里的每个三字母组合需要花半微秒的时间。

每个三字母组合的插入时间在ETS有序异键表里是2.9微秒，在ETS异键表里是1.5微秒，在Erlang异键表则里是9.3微秒。就存储而言，每个三字母组合在ETS异键表和有序异键表里都占用19个字节，而在sets模块里是9个字节。

19.4.4　访问表所需的时间

好吧，看来创建这些表需要一定的时间，但它不是这个案例的重点。现在编写一些代码来测量访问时间。我们将对表里的每一个三字母组合各查询一次，然后得出平均查询时间。这是执行计时的代码：

lib_trigrams.erl

```
timer_tests() ->
    time_lookup_ets_set("Ets ordered Set", "trigramsOS.tab"),
    time_lookup_ets_set("Ets set", "trigramsS.tab"),
    time_lookup_module_sets().

time_lookup_ets_set(Type, File) ->
```

```
    {ok, Tab} = ets:file2tab(File),
    L = ets:tab2list(Tab),
    Size = length(L),
    {M, _} = timer:tc(?MODULE, lookup_all_ets, [Tab, L]),
    io:format("~s lookup=~p micro seconds~n",[Type, M/Size]),
    ets:delete(Tab).

lookup_all_ets(Tab, L) ->
    lists:foreach(fun({K}) -> ets:lookup(Tab, K) end, L).

time_lookup_module_sets() ->
    {ok, Bin} = file:read_file("trigrams.set"),
    Set = binary_to_term(Bin),
    Keys = sets:to_list(Set),
    Size = length(Keys),
    {M, _} = timer:tc(?MODULE, lookup_all_set, [Set, Keys]),
    io:format("Module set lookup=~p micro seconds~n",[M/Size]).

lookup_all_set(Set, L) ->
    lists:foreach(fun(Key) -> sets:is_element(Key, Set) end, L).
```

19

现在我们开始：

```
1> lib_trigrams:timer_tests().
Ets ordered Set lookup=1.79964 micro seconds
Ets set lookup=0.719279 micro seconds
Module sets lookup=1.35268 micro seconds
ok
```

这些时间是平均每次查询所需的微秒数。

19.4.5 获胜者是……

结果是压倒性的，ETS异键表以相当大的优势取得了胜利。在我的机器上，sets每次查询耗时半微秒，这是非常好的成绩！

注意 执行前面这样的测试来实际测量某个操作的耗时是一种良好的编程实践。无需将此做到极致来把所有耗时都测试一遍，比如给所有操作计时，只需针对程序里那些最耗时的操作即可。那些不太耗时的操作应该尽量用最优美的方式进行编写。如果我们因为效率的原因被迫编写一些不直观的丑陋代码，就应该加上详细的文档。

现在可以编写预测字符串是否是正规英语单词的函数了。

为了测试某个字符串是否可能是英语单词，我们将遍历字符串里的所有三字母组合，检查它们是否包含在之前计算出的三字母组合表里。is_word函数做的就是这件事。

lib_trigrams.erl

```
is_word(Tab, Str) -> is_word1(Tab, "\s" ++ Str ++ "\s").
is_word1(Tab, [_,_,_]=X) -> is_this_a_trigram(Tab, X);
is_word1(Tab, [A,B,C|D]) ->
    case is_this_a_trigram(Tab, [A,B,C]) of
        true  -> is_word1(Tab, [B,C|D]);
        false -> false
    end;
is_word1(_, _) ->
    false.
is_this_a_trigram(Tab, X) ->
    case ets:lookup(Tab, list_to_binary(X)) of
        [] -> false;
        _  -> true
    end.
open() ->
    File = filename:join(filename:dirname(code:which(?MODULE)),
                         "/trigramsS.tab"),
    {ok, Tab} = ets:file2tab(File),
    Tab.
close(Tab) -> ets:delete(Tab).
```

函数`open`会打开我们之前计算出的ETS表，它必须在`is_word`之前调用。

此处用到的另一个技巧，正是我定位包含三字母组合表的文件时的做法，就是把它保存在与当前模块代码相同的目录里。`code:which(?MODULE)`会返回`?MODULE`目标代码所属文件的名称。

19.5　保存元组到磁盘

ETS表把元组保存在内存里，而DETS提供了把Erlang元组保存到磁盘上的方法。DETS的最大文件大小是2GB。DETS文件必须先打开才能使用，用完后还应该正确关闭。如果没有正确关闭，它们就会在下次打开时自动进行修复。因为修复可能会花很长一段时间，所以先正确关闭它们再结束程序是很重要的。

DETS表有着和ETS表不同的共享属性。DETS表在打开时必须赋予一个全局名称。如果两个或更多本地进程用相同的名称和选项打开某个DETS表，它们就会共享这个表。这个表会一直处于打开状态，直到所有进程都关闭它（或者崩溃）。

范例：文件名索引

创建一个基于磁盘的表，它将把文件名映射到整数上，反之亦然。我们将定义函数`filename2index`和它的逆函数`index2filename`。

为了实现这个索引，我们将创建一个DETS表，然后用三种不同类型的元组填充它。

- `{free, N}`
  `N`是表里的第一个空白索引。当我们在表里输入一个新文件名时，就会指派索引`N`给它。

- {FileNameBin, K}
  FileNameBin（一个二进制型）被指派了索引K。
- {K, FileNameBin}
  K（一个整数）代表文件FilenameBin。

请注意，每个新添加的文件都会在表里增加两个条目：一个File → Index条目和一个逆向的Index → Filename条目。这是出于效率的原因。当ETS或DETS表建立时，元组里只有一个元素充当键的角色。匹配某个非键元组元素虽然可行，但非常低效，因为它涉及对整个表进行搜索。当整个表都在磁盘上时，这个操作的代价尤其高昂。

现在来编写这个程序。首先编写打开和关闭DETS表的函数，这个表用来保存所有的文件名。

lib_filenames_dets.erl

```
-module(lib_filenames_dets).
-export([open/1, close/0, test/0, filename2index/1, index2filename/1]).

open(File) ->
    io:format("dets opened:~p~n", [File]),
    Bool = filelib:is_file(File),
    case dets:open_file(?MODULE, [{file, File}]) of
        {ok, ?MODULE} ->
            case Bool of
                true  -> void;
                false -> ok = dets:insert(?MODULE, {free,1})
            end,
            true;
        {error,Reason} ->
            io:format("cannot open dets table~n"),
            exit({eDetsOpen, File, Reason})
    end.

close() -> dets:close(?MODULE).
```

open的代码会自动初始化DETS表，方法是在创建新表时插入{free, 1}元组。如果File存在，filelib:is_file(File)就会返回true，否则返回false。请注意，dets:open_file或者创建一个新文件，或者打开一个现有文件，这就是必须在调用dets:open_file前先检查文件是否存在的原因。

我们在这段代码里多次使用了?MODULE宏，它会展开成当前的模块名（即lib_filenames_dets）。许多针对DETS的函数调用需要一个唯一的原子参数作为表名。单凭模块名就能生成一个唯一的表名。因为系统里不能有两个名称相同的Erlang模块，所以如果处处遵循这一惯例，就有理由认定我们有一个可以用于表名的唯一名称。

我每次都用?MODULE宏而不是显式指定模块名，这是因为我有一个习惯，就是会在编写代码的过程中修改模块名。用了宏之后，即使修改了模块名，代码也会是正确的。

打开文件之后，向表里插入一个新文件名就很简单了。它是由filename2index调用产生的

副作用实现的。如果文件名在表里，就会返回它的索引，否则就会生成一个新索引并更新表，而这次更新会使用三个元组。

lib_filenames_dets.erl

```
filename2index(FileName) when is_binary(FileName) ->
    case dets:lookup(?MODULE, FileName) of
        [] ->
            [{_,Free}] = dets:lookup(?MODULE, free),
            ok = dets:insert(?MODULE,
                   [{Free,FileName},{FileName,Free},{free,Free+1}]),
            Free;
        [{_,N}] ->
            N
    end.
```

请注意我们是如何把三个元组保存到表里的。dets:insert的第二个参数既可以是一个元组，也可以是一个元组列表。另外要注意的是文件名用一个二进制型表示，这是出于效率的原因。建议你养成在ETS和DETS表里用二进制型来表示字符串的习惯。

细心的读者也许会注意到filename2index里有一处潜在的竞争状况。如果在dets:insert尚未被调用时有两个并行进程调用了dets:lookup，filename2index就会返回一个不正确的值。为了让这个函数能正常工作，必须确保任一时刻都只能有一个进程调用它。

把索引转换成文件名非常简单。

lib_filenames_dets.erl

```
index2filename(Index) when is_integer(Index) ->
    case dets:lookup(?MODULE, Index) of
        []        -> error;
        [{_,Bin}] -> Bin
    end.
```

这里做了一个小小的设计决定：调用index2filename(Index)时如果没有文件名关联这个索引该怎么办。可以调用exit(ebadIndex)来让调用进程崩溃，但在这里我们选择了一种更温和的方式：只返回原子error。调用进程能够分辨出合法文件名和不正确的值，因为所有合法返回的文件名都是二进制型。

另外还需要注意filename2index和index2filename里的关卡测试。它们检查参数是否为所要求的类型。做这些测试是一个好主意，因为把错误类型的数据输入DETS表可能会造成非常难以调试的情况。可以想象一下，如果把错误类型的数据保存在表里然后过几个月再读取它，那时做什么都已经太晚了。最好的做法是先检查所有数据的正确性，然后再把它们添加到表里。

19.6　其余操作

ETS和DETS表还支持许多尚未在本章里介绍过的操作。这些操作分为以下几类：

- 基于模式获取和删除对象；
- ETS和DETS表之间，以及ETS表和磁盘文件之间的相互转换；
- 查看表的资源占用情况；
- 遍历表内所有元素；
- 修复损坏的DETS表；
- 让表可视化。

网上可以找到更多关于ETS[1]和DETS[2]的信息。

ETS和DETS表被设计用来对Erlang数据进行高效的底层内存和磁盘存储，但它们并不是终极解决方案。对于更复杂的数据存储而言，我们需要一种数据库。

在下一章里，我们将介绍Mnesia，它是一种用Erlang编写的实时数据库，也是标准Erlang分发套装的一部分。Mnesia的内部使用ETS和DETS表，而且`ets`和`dets`模块导出的函数中有许多是供Mnesia内部使用的。Mnesia能做各种用单个ETS和DETS表无法实现的操作。例如，可以索引主键以外的元素，因此我们在`filename2index`示例里使用的两次插入技巧就不再是必需的了。Mnesia实际上会创建多个ETS或DETS表来做到这一点，但这些对用户是隐藏的。

19

19.7 练习

(1) `Mod:module_info(exports)`会返回`Mod`模块里所有导出函数的列表。用这个函数找出Erlang系统库里导出的所有函数。制作一个键–值查询表，其中键是一个`{Function,Arity}`对，值是一个模块名。把这些数据储存在ETS和DETS表里。

> **提示** 使用`code:lib_dir()`和`code:lib_dir(LibName)`来找出系统里所有模块的名称。

(2) 制作一个共享的ETS计数表。实现一个名为`count:me(Mod,Line)`的函数，通过在你的代码里添加`count:me(?MODULE, ?LINE)`来调用它。每当这个函数被调用时，就给记录自身执行次数的计数器加1。编写一些函数来初始化和读取计数器。

(3) 编写一个检测文本抄袭的程序。用一个双遍历（two-pass）算法来实现它。第一次遍历时，把文本打散成40个字符的小块并计算各个块的校验和，然后把校验和与文件名保存在一个ETS表里。第二次遍历时，计算数据里各个40字符块的校验和，并把它们与ETS表里的校验和进行比较。

> **提示** 要做到这一点，需要计算“滚动校验和”[3]。举个例子，假如`C1 = B1 + B2 + ... B40`并且`C2 = B2 + B3 + ... B41`，你就可以快速计算出`C2`，因为通过观察能发现`C2 = C1 + B41 – B1`。

① http://www.erlang.org/doc/man/ets.html
② http://www.erlang.org/doc/man/dets.html
③ http://en.wikipedia.org/wiki/Rolling_hash

第 20 章

Mnesia：Erlang数据库

要编写一个多用户游戏、制作一个新网站，或者创建一个在线支付系统，多半需要一个数据库管理系统（Database Management System，简称DBMS）。

Mnesia是一种用Erlang编写的数据库，它被用于高要求的电信应用程序，同时也是标准Erlang分发套装的一部分。将它配置为内存复制后，就能在两个物理隔离的节点上实现快速的容错式数据存储。它还支持事务，并有自己的查询语言。

Mnesia的速度极快，可以保存任何类型的Erlang数据结构。它还是高度可定制的。数据表既可以保存在内存里（为了速度），也可以保存在磁盘上（为了持久性）。表还可以在不同机器之间进行复制，从而实现容错行为。

20.1　创建初始数据库

在做任何事之前，必须先创建一个Mnesia数据库。这件事只需要做一次。

```
$ erl
1> mnesia:create_schema([node()]).
ok
2> init:stop().
ok
$ ls
Mnesia.nonode@nohost
```

`mnesia:create_schema(NodeList)`会在`NodeList`（它必须是一个包含有效Erlang节点的列表）里的所有节点上都初始化一个新的Mnesia数据库。这个案例给出的节点列表是`[node()]`，也就是当前节点。Mnesia完成初始化并创建了一个名为`Mnesia.nonode@nohost`的目录结构来保存数据库。

为什么这个DBMS被称为Mnesia

它最初的名字是Amnesia（健忘症）。我们的某个老板不喜欢这个名字，他说：“不能叫它Amnesia，没人想要一个会忘事的数据库！”所以我们去掉了A，这个名字就留了下来。

然后退出Erlang shell，用操作系统的ls命令进行验证。

如果在一个名为joe的分布式节点上重复这个练习，就会得到以下输出。

```
$ erl -name joe
(joe@doris.myerl.example.com) 1> mnesia:create_schema([node()]).
ok
(joe@doris.myerl.example.com) 2> init:stop().
ok
$ ls
Mnesia.joe@doris.myerl.example.com
```

也可以在启动Erlang时指向一个特定的数据库。

```
$ erl  -mnesia dir '"/home/joe/some/path/to/Mnesia.company"'
1> mnesia:create_schema([node()]).
ok
2> init:stop().
ok
```

/home/joe/some/path/to/Mnesia.company是将要保存这个数据库的目录。

20.2 数据库查询

创建完数据库之后，我们就可以拿它练练手了。首先来看Mnesia的查询。浏览它们之后，你也许会惊讶地发现Mnesia的查询非常像SQL和列表推导，所以实际上你不需要学习什么就能上手。事实上，列表推导和SQL之间的高度相似性并不是什么奇怪的事情，因为它们都是基于数学里的集合论。

我在所有的例子里都假定你已经创建了一个包含shop和cost两个表的数据库。这些表包含的数据如表8和表9所示。

表8 shop表

商　品	数　量	费　用
apple	20	2.3
orange	100	3.8
pear	200	3.6
banana	420	4.5
potato	2456	1.2

表9 cost表

名　称	价　格
apple	1.5
orange	2.4
pear	2.2
banana	1.5
potato	0.6

Mnesia里的表是一个包含若干行的异键或同键表，其中每一行都是一个Erlang记录。要在Mnesia里表示这些表，需要一些记录定义来对表里的行进行定义，代码如下：

test_mnesia.erl

```
-record(shop, {item, quantity, cost}).
-record(cost, {name, price}).
```

在操作数据库之前，需要创建一个数据库架构（schema），启动数据库，添加一些表定义并停止数据库，然后再重启它。这些事只需要做一遍。下面是代码：

test_mnesia.erl

```
do_this_once() ->
    mnesia:create_schema([node()]),
    mnesia:start(),
    mnesia:create_table(shop,   [{attributes, record_info(fields, shop)}]),
    mnesia:create_table(cost,   [{attributes, record_info(fields, cost)}]),
    mnesia:create_table(design, [{attributes, record_info(fields, design)}]),
    mnesia:stop().
```

```
1> test_mnesia:do_this_once().
stopped
=INFO REPORT==== 24-May-2013::15:27:56 ===
application: mnesia
exited: stopped
type: temporary
```

接下来看一下具体示例。

20.2.1　选择表里的所有数据

下面是选择shop表里所有数据的代码。（对那些懂SQL的读者，我们在代码里用注释符号开头的片段展示了能执行相应操作的等效SQL命令。）

test_mnesia.erl

```
%% 等效SQL命令
%%  SELECT * FROM shop;

demo(select_shop) ->
    do(qlc:q([X || X <- mnesia:table(shop)]));
```

这段代码的重点是`qlc:q`调用，它会把查询（也就是它的参数）编译成一种用于查询数据库的内部格式。把编译后的查询传递给一个名为`do()`的函数，它会在接近`test_mnesia`底部的位置进行定义，负责运行查询并返回结果。为了让这一切能够轻易从`erl`里调用，我们把它映射到函数`demo(select_shop)`上。（整个模块位于`code/test_mnesia.erl`文件内，可以在本书的主页[①]里找到。）

① http://pragprog.com/titles/jaerlang2/source_code

在使用这个数据库之前，需要做一些例行的启动和载入表定义工作。它们必须在使用数据库之前运行，但在每个Erlang会话里只需运行一次。

test_mnesia.erl

```
%% 等级SQL命令
%%  SELECT * FROM shop;

demo(select_shop) ->
    do(qlc:q([X || X <- mnesia:table(shop)]));
```

现在可以启动数据库并生成一个查询了。

```
1> test_mnesia:start().
ok
2> test_mnesia:reset_tables().
{atomic, ok}
3> test_mnesia:demo(select_shop).
[{shop,potato,2456,1.2},
{shop,orange,100,3.8},
{shop,apple,20,2.3},
{shop,pear,200,3.6},
{shop,banana,420,4.5}]
```

注意　表里各行出现的顺序是随机的。

在这个示例中构建查询的代码行如下：

```
qlc:q([X || X <- mnesia:table(shop)])
```

它看上去非常像一个列表推导（参见4.5节）。事实上，`qlc`就代表了query list comprehension（查询列表推导）。它是其中一个可以用来访问Mnesia数据库内数据的模块。

`[X || X <- mnesia:table(shop)]`的意思是“一个由X组成的列表，X提取自`shop`这个Mnesia表”。X的值是Erlang的`shop`记录。

注意　`qlc:q/1`的参数必须是一个字面上的列表推导，不能是通过求值得出的。举个例子，下面的代码与示例里的代码不是等价的。

```
Var = [X || X <- mnesia:table(shop)],
qlc:q(Var)
```

20.2.2　从表里选择数据

下面这个查询会从`shop`表里选择`item`和`quantity`列：

test_mnesia.erl

```
%% 等效SQL命令
%%  SELECT item, quantity FROM shop;

demo(select_some) ->
    do(qlc:q([{X#shop.item, X#shop.quantity} || X <- mnesia:table(shop)]));
```

```
4> test_mnesia:demo(select_some).
[{orange,100},{pear,200},{banana,420},{potato,2456},{apple,20}]
```

上面这个查询里的X值是类型为shop的记录。回想一下5.2节里介绍过的记录语法，你就会记得X#shop.item指的是shop记录里的item字段。因此，元组{X#shop.item, X#shop. quantity}是一个由X的item和quantity字段所组成的元组。

20.2.3　从表里有条件选择数据

下面这个查询会列出shop表里所有符合条件（库存数量小于250）的商品。也许可以用它来确定哪些商需要再次订购。请注意这个条件如何自然地表达为列表推导的一部分。

test_mnesia.erl

```
%% 等效SQL命令
%%   SELECT shop.item FROM shop
%%   WHERE  shop.quantity < 250;

demo(reorder) ->
    do(qlc:q([X#shop.item || X <- mnesia:table(shop),
                             X#shop.quantity < 250
                                ]));
```

```
5> test_mnesia:demo(reorder).
[orange,pear,apple]
```

20.2.4　从两个表里选择数据（联接）

现在假设想要重新订购的商品必须库存少于250，并且价格低于2.0个货币单位。要做到这一点，我们需要访问两个表。这个查询如下：

test_mnesia.erl

```
%% 等效SQL命令
%%   SELECT shop.item
%%   FROM shop, cost
%%   WHERE shop.item = cost.name
%%     AND cost.price < 2
%%     AND shop.quantity < 250

demo(join) ->
    do(qlc:q([X#shop.item || X <- mnesia:table(shop),
                             X#shop.quantity < 250,
```

```
                            Y <- mnesia:table(cost),
                            X#shop.item =:= Y#cost.name,
                            Y#cost.price < 2
                              ])).
```

```
6> test_mnesia:demo(join).
[apple]
```

这里的关键在于联接shop表里的item名称和cost表里的name。

```
X#shop.item =:= Y#cost.name
```

20.3　添加和移除数据库里的数据

我们再次假定你已经创建了数据库并定义了一个shop表。现在我们想为该表添加或移除行。

20.3.1　添加行

可以像下面这样给shop表添加一行：

test_mnesia.erl

```
add_shop_item(Name, Quantity, Cost) ->
    Row = #shop{item=Name, quantity=Quantity, cost=Cost},
    F = fun() ->
                mnesia:write(Row)
        end,
    mnesia:transaction(F).
```

它会创建一个shop记录并把它插入表中。

```
1> test_mnesia:start().
ok
2> test_mnesia:reset_tables().
{atomic, ok}
%% 列出shop表
3> test_mnesia:demo(select_shop).
[{shop,orange,100,3.80000},
 {shop,pear,200,3.60000},
 {shop,banana,420,4.50000},
 {shop,potato,2456,1.20000},
 {shop,apple,20,2.30000}]
%% 添加一个新行
4> test_mnesia:add_shop_item(orange, 236, 2.8).
 {atomic,ok}
%% 再次列出shop表，看看有什么变化
5> test_mnesia:demo(select_shop).
[{shop,orange,236,2.80000},
 {shop,pear,200,3.60000},
 {shop,banana,420,4.50000},
 {shop,potato,2456,1.20000},
 {shop,apple,20,2.30000}]
```

注意 shop表的主键是表内的第一列，也就是shop记录里的item字段。这个表属于“异键”类型（参见19.1节里对异键和同键类型的讨论）。如果新创建的记录和数据表里的某一行具有相同的主键，就会覆盖那一行，否则就会创建一个新行。

20.3.2　移除行

要移除某一行，需要知道该行的对象ID（Object ID，简称OID）。它由表名和主键的值构成。

test_mnesia.erl

```
remove_shop_item(Item) ->
    Oid = {shop, Item},
    F = fun() ->
                mnesia:delete(Oid)
        end,
    mnesia:transaction(F).
```

```
6> test_mnesia:remove_shop_item(pear).
{atomic,ok}
%% 列出这个表，pear已经不见了
7> test_mnesia:demo(select_shop).
[{shop,orange,236,2.80000},
 {shop,banana,420,4.50000},
 {shop,potato,2456,1.20000},
 {shop,apple,20,2.30000}]
8> mnesia:stop().
ok
```

20.4　Mnesia 事务

之前向数据库添加或移除数据以及进行查询时，我们编写了如下代码：

```
do_something(...) ->
    F = fun() ->
            % ...
            mnesia:write(Row)
            % ... 或者 ...
            mnesia:delete(Oid)
            % ... 或者 ...
            qlc:e(Q)
        end,
    mnesia:transaction(F)
```

F是一个不带参数的fun。我们在F里调用了下列函数的某种组合形式：mnesia:write/1、mnesia:delete/1和qlc:e(Q)（Q是用qlc:q/1编译的查询）。构建完fun后，调用mnesia:transaction(F)，它会执行fun里的表达式序列。

事务（transaction）能阻止有缺陷的程序代码，但更重要的是，它能阻止对数据库的并发访

问。假设有两个进程试图同时访问相同的数据。比方说，假设我的银行账户里有10美元，有两个人试图同时从这个账户里取出8美元。我希望让其中一笔交易（即事务）成功，而另一笔失败。

这正是mnesia:transaction/1所提供的保证。某个事务里对数据表的读写操作或者都成功，或者都不成功。如果都不成功，我们就会说这个事务失败了。如果事务失败了，就不会对数据库做任何改动。

Mnesia采用一种悲观锁定（pessimistic locking）的策略。每当Mnesia事务管理器访问一个表时，都会根据上下文情况尝试锁定记录甚至整个表。如果它发现这可能导致死锁，就会立即中止事务并撤销之前所做的改动。

如果因为其他进程正在访问数据而导致事务一开始就失败了，系统就会进行短时间的等待，然后再次尝试执行事务。这么做的一种结果就是事务fun里的代码可能会被执行很多次。

出于这个原因，事务fun里的代码不应该做任何带有副作用的事情。举个例子，如果打算编写以下代码：

```
F = fun() ->
        ...
        io:format("reading ..."), %% don't do this
        ...
    end,
mnesia:transaction(F),
```

也许就会得到大量输出，因为这个fun可能会被多次重试。

注意1 对mnesia:write/1和mnesia:delete/1的调用只应该出现在由mnesia:transaction/1处理的fun内部。

注意2 永远不要编写代码来显式捕捉Mnesia访问函数（mnesia:write/1和mnesia: delete/1等）里的异常错误，因为Mnesia的事务机制本身就依赖这些函数在失败时抛出异常错误。如果你捕捉这些异常错误并试图自行处理它们，就会破坏事务机制。

20.4.1 中止事务

我们的商店附近有一个农场，那里有农民在种植苹果。这个农民很喜欢橙子，他用苹果来交易橙子。当前的价格是两个苹果换一个橙子。因此，要购买N个橙子，农民需要支付2*N个苹果。下面的函数会在农民购买橙子时更新数据库：

test_mnesia.erl

```
farmer(Nwant) ->
    %% Nwant = 农民想要购买的橙子数量
    F = fun() ->
                %% 找出苹果的数量
                [Apple] = mnesia:read({shop,apple}),
                Napples = Apple#shop.quantity,
                Apple1  = Apple#shop{quantity = Napples + 2*Nwant},
                %% 更新数据库
                mnesia:write(Apple1),
```

```
                %% 找出橙子的数量
                [Orange] = mnesia:read({shop,orange}),
                NOranges = Orange#shop.quantity,
                if
                    NOranges >= Nwant ->
                        N1 =  NOranges - Nwant,
                        Orange1 = Orange#shop{quantity=N1},
                        %% 更新数据库
                        mnesia:write(Orange1);
                    true ->
                        %% 糟糕，橙子数量不够
                        mnesia:abort(oranges)
                end
        end,
    mnesia:transaction(F).
```

这段代码是用一种相当笨拙的方式编写的，因为我想展示事务机制是如何工作的。首先，我更新了数据库里的苹果数量。这发生在我检查橙子的数量之前。这么做是为了显示如果事务失败它就会被撤销。通常情况下，我会暂缓把橙子和苹果数据写回数据库，直到我确定自己有足够的橙子。

让我们来展示一下它的实际工作情况。早上，农民过来买了50个橙子。

```
1> test_mnesia:start().
ok
2> test_mnesia:reset_tables().
{atomic, ok}
%% 列出shop表
3> test_mnesia:demo(select_shop).
[{shop,orange,100,3.80000},
 {shop,pear,200,3.60000},
 {shop,banana,420,4.50000},
 {shop,potato,2456,1.20000},
 {shop,apple,20,2.30000}]
%% 农民买了50个橙子
%% 支付了100个苹果
4> test_mnesia:farmer(50).
{atomic,ok}
%% 再次打印shop表
5> test_mnesia:demo(select_shop).
[{shop,orange,50,3.80000},
 {shop,pear,200,3.60000},
 {shop,banana,420,4.50000},
 {shop,potato,2456,1.20000},
 {shop,apple,120,2.30000}]
```

下午，农民想要再买100个橙子（这家伙是有多爱橙子啊）。

```
6> test_mnesia:farmer(100).
{aborted,oranges}
```

```
7> test_mnesia:demo(select_shop).
[{shop,orange,50,3.80000},
 {shop,pear,200,3.60000},
 {shop,banana,420,4.50000},
 {shop,potato,2456,1.20000},
 {shop,apple,120,2.30000}]
```

当这个事务失败时（即调用mnesia:abort(Reason)时），mnesia:write所做的更改都被撤销了。正因为如此，数据库恢复到了进入事务之前的状态。

20.4.2 载入测试数据

现在我们已经了解了事务的工作原理，下面可以来看看载入测试数据的代码了。

test_mnesia:example_tables/0函数的作用是提供数据来初始化各个数据表。元组的第一个元素是表名，后面是遵循原始记录定义顺序的表数据。

test_mnesia.erl

```
example_tables() ->
    [%% shop表
     {shop, apple,   20,   2.3},
     {shop, orange,  100,  3.8},
     {shop, pear,    200,  3.6},
     {shop, banana,  420,  4.5},
     {shop, potato,  2456, 1.2},
     %% cost表
     {cost, apple,   1.5},
     {cost, orange,  2.4},
     {cost, pear,    2.2},
     {cost, banana,  1.5},
     {cost, potato,  0.6}
    ].
```


接下来是负责把来自示例表的数据插入Mnesia的代码。它所做的就是对example_tables/1返回列表里的每一个元组调用mnesia:write。

test_mnesia.erl

```
reset_tables() ->
    mnesia:clear_table(shop),
    mnesia:clear_table(cost),
    F = fun() ->
                foreach(fun mnesia:write/1, example_tables())
        end,
    mnesia:transaction(F).
```

20.4.3 do()函数

demo/1调用的do()函数略微复杂一些。

test_mnesia.erl

```
do(Q) ->
    F = fun() -> qlc:e(Q) end,
    {atomic, Val} = mnesia:transaction(F),
    Val.
```

它在一个Mnesia事务内调用了`qlc:e(Q)`。`Q`是一个已编译的QLC查询，而`qlc:e(Q)`会执行这个查询，并把查询到的所有结果以列表的形式返回。返回值`{atomic, Val}`的意思是事务成功并得到了`Val`值。`Val`是这个事务函数的值。

20.5　在表里保存复杂数据

传统DBMS的缺点之一是在列里保存的数据类型数量有限。你可以保存一个整数，一个字符串，一个浮点数，诸如此类。但如果想要保存一个复杂的对象，就会有麻烦了。举个例子，如果你是一个Java程序员，在SQL数据库里保存一个Java对象就是非常麻烦的。

Mnesia被设计用来保存Erlang的数据结构。事实上，可以在Mnesia表里保存任何你喜欢的Erlang数据结构。

为了演示这一点，我们假定有许多建筑师想要把他们的设计保存在一个Mnesia数据库里。首先必须定义一个记录来表示他们的设计。

test_mnesia.erl

```
-record(design, {id, plan}).
```

然后定义一个给数据库添加设计的函数。

test_mnesia.erl

```
add_plans() ->
    D1 = #design{id   = {joe,1},
                 plan = {circle,10}},
    D2 = #design{id   = fred,
                 plan = {rectangle,10,5}},
    D3 = #design{id   = {jane,{house,23}},
                 plan = {house,
                         [{floor,1,
                           [{doors,3},
                            {windows,12},
                            {rooms,5}]},
                          {floor,2,
                           [{doors,2},
                            {rooms,4},
                            {windows,15}]}]}},
    F = fun() ->
                mnesia:write(D1),
                mnesia:write(D2),
                mnesia:write(D3)
```

```
        end,
    mnesia:transaction(F).
```

现在可以给数据库添加一些设计了。

```
1> test_mnesia:start().
ok
2> test_mnesia:add_plans().
{atomic,ok}
```

现在我们的数据库里有了一些设计方案。可以用下面这个访问函数来提取它们：

test_mnesia.erl

```
get_plan(PlanId) ->
    F = fun() -> mnesia:read({design, PlanId}) end,
    mnesia:transaction(F).
```

```
3> test_mnesia:get_plan(fred).
{atomic,[{design,fred,{rectangle,10,5}}]}
4> test_mnesia:get_plan({jane, {house,23}}).
{atomic,[{design,{jane,{house,23}},
                 {house,[{floor,1,[{doors,3},
                                   {windows,12},
                                   {rooms,5}]},
                         {floor,2,[{doors,2},
                                   {rooms,4},
                                    {windows,15}]}]}}]}
```

如你所见，数据库的键和提取出来的记录都可以是任意的Erlang数据类型。

用专业术语来说，数据库里的数据结构和编程语言里的数据结构之间不存在*阻抗失配*（impedance mismatch）。这就意味着向数据库插入和删除复杂的数据结构是非常快速的。

20.6　表的类型和位置

可以对Mnesia表进行多种方式的配置。首先，表可以位于内存或磁盘里（或者两者皆有）。其次，表可以位于单台机器上，也可以在多台机器之间复制。

在设计表时必须考虑到想要保存在表里的数据类型。以下是各类表的性质。

- *内存表*

  它们的速度非常快，但是里面的数据是*易失*的，所以如果机器崩溃或者你停止了DBMS，数据就会丢失。

- *磁盘表*

  磁盘表应该不会受到系统崩溃的影响（前提是磁盘没有物理损坏）。

  当Mnesia事务写入一个表并且这个表是保存在磁盘上时，实际上是事务数据首先被写入了一个磁盘日志。这个磁盘日志会不断增长，里面的信息会每隔一段时间与数据库里的其他数据合并，然后磁盘日志里的条目就会被清除。如果系统崩溃了，磁盘日志就会在

下一次系统重启时进行一致性检查，任何未合并的日志条目会先添加到数据库里，然后数据库才可用。任何一个事务成功时，数据都应该已经正确写入到磁盘日志里，如果系统随后崩溃了，那么当它下次重启时，事务所做的改动应该会完好无损。

如果系统在事务进行过程中崩溃了，那么它对数据库所做的改动应该会丢失。

使用内存表之前，需要做一些试验来看看物理内存是否能容纳整个表。如果物理内存装不下内存表，系统就会频繁读写页面文件，这将会影响性能。

内存表是易失的，所以如果想构建一个容错式应用程序，就需要把内存表复制到磁盘上，或者把它复制为第二台机器的内存或磁盘表，或者两者皆有。

分段表

Mnesia支持“分段”的表（用数据库的行话说就是水平分区）。它被设计用来实现极其巨大的表。这些表被分成许多片段，然后在不同的机器上储存。这些片段自身都是Mnesia表，就像任何其他的表一样，它们可以被复制，可以有索引，诸如此类。

更多细节请参阅Mnesia用户指南。

20.6.1　创建表

要创建一个表，我们会调用`mnesia:create_table(Name, ArgS)`，其中`ArgS`是一个由`{Key,Val}`元组构成的列表。如果表创建成功，`create_table`就会返回`{atomic, ok}`，否则返回`{aborted, Reason}`。下面是`create_table`最常用的一些参数。

- `Name`
  它是表的名称（一个原子）。按惯例它是一个Erlang记录的名称，表里的各行是这个记录的实例。
- `{type, Type}`
  它指定了表的类型。`Type`是`set`、`ordered_set`或`bag`中的一个。这些类型与19.1节里描述的类型含义相同。
- `{disc_copies, NodeList}`
  `NodeList`是一个Erlang节点列表，这些节点将保存表的磁盘副本。当我们使用这个选项时，系统还会在执行这个操作的节点上创建一个表的内存副本。
  你可以既在一个节点上保存`disc_copies`类型的副本表，又在另一个节点上保存该表的不同类型。这种做法能满足以下要求：
  - 读取操作非常快，并在内存里执行；
  - 写入操作在持久性存储介质里执行。
- `{ram_copies, NodeList}`
  `NodeList`是一个Erlang节点列表，这些节点将保存表的内存副本。

- {disc_only_copies, NodeList}
  NodeList是一个Erlang节点列表，这些节点将只保存表的磁盘副本。这些表没有内存副本，访问起来会比较慢。
- {attributes, AtomList}
  这个列表包含表里各个值的列名。请注意，要创建一个包含Erlang记录xxx的表，可以用{attributes, record_info(fields, xxx)}这种语法（也可以显式指定一个记录字段名列表）。

注意　create_table的选项比我在这里展示的更多。所有选项的细节请参阅mnesia的手册页。

20.6.2 常用的表属性组合

假定下面所有的Attrs都代表一个{attributes,...}元组。

这里是一些涵盖了大多数常见情形的表配置选项。

- mnesia:create_table(shop, [Attrs])
- ❑ 它会在单个节点上创建一个常驻内存的表。
- ❑ 如果节点崩溃了，表就会丢失。
- ❑ 它是所有表里最快的一种。
- ❑ 内存必须能容纳这个表。

- mnesia:create_table(shop,[Attrs,{disc_copies,[**node**()]}])
- ❑ 它会在单个节点上创建一个常驻内存的表和一个磁盘副本。
- ❑ 如果节点崩溃了，表就会从磁盘恢复。
- ❑ 表的读访问很快，但写访问较慢。
- ❑ 内存最好能容纳这个表。

- mnesia:create_table(shop, [Attrs,{disc_only_copies,[**node**()]}])
- ❑ 它只会在单个节点上创建一个磁盘副本。
- ❑ 它用于那些因为太大而无法放入内存的表。
- ❑ 它的访问速度比带有内存副本的方案更慢。

- mnesia:create_table(shop,
  [Attrs,{ram_copies,[**node**(),someOtherNode()]})
- ❑ 它会在两个节点上各创建一个常驻内存的表。
- ❑ 如果两个节点都崩溃了，表就会丢失。
- ❑ 内存必须能容纳这个表。
- ❑ 可以在任何一个节点上访问这个表。

- `mnesia:create_table(shop,`
  `[Attrs, {disc_copies, [node(),someOtherNode()]}])`
  - ❑ 它会在多个节点上创建磁盘副本。
  - ❑ 无论哪个节点崩溃，我们都能恢复过来。
  - ❑ 即使所有节点都崩溃了，表也不会丢失。

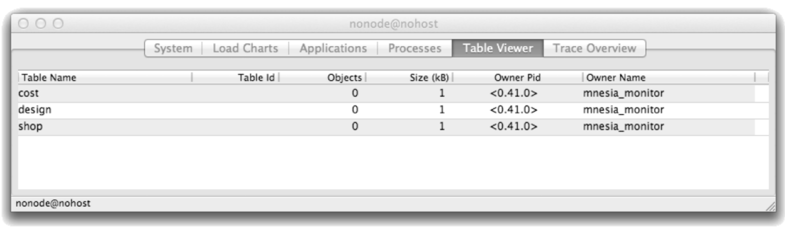
20.6.3 表的行为

当一个表被复制到多个Erlang节点时，它会尽可能远地进行同步。如果某个节点崩溃了，系统仍然会正常工作，但是副本的数量会减少。当崩溃的节点重新上线时，它会与其他存有副本的节点重新进行同步。

注意 如果运行Mnesia的节点停止工作，Mnesia就可能会过载。如果你使用的笔记本电脑进入了睡眠，那么当它被唤醒时，Mnesia就可能会暂时处于过载状态，并生成许多警告消息。我们可以忽略这些消息。

20.7 表查看器

要查看我们保存在Mnesia里的数据，可以使用“observer”应用程序里内建的表查看器。先用命令`observer:start()`启动observer，然后点击“Table Viewer”（表查看器）标签。现在在observer的控制菜单里选择“View”（查看）>“Mnesia Tables”（Mnesia表）。你会看到一个表的清单，如下图所示：



点击shop条目会打开一个新窗口。



通过使用observer，你可以查看表，检查系统状态，查看进程或执行其他一些操作。

20.8 深入挖掘

希望现在你对Mnesia兴趣大增了。Mnesia是一种非常强大的DBMS。它从1998年开始就被用于生产环境，比如爱立信公司推出的许多高要求电信应用程序。

因为本书是关于Erlang而不是Mnesia的，所以我只能给出几种最常见的Mnesia用例。我在这一章里展示的技巧都是我自己用过的。我实际使用（或理解）的并不比这里展示的多多少。但是通过向你展示的这些内容，你可以找到很多乐子，并能制作出一些相当复杂的应用程序。

我主要省略了以下这些主题。

- 备份和恢复

  Mnesia有一系列的选项用于配置备份操作，可以实现不同类型的灾难恢复。

- 脏操作

  Mnesia允许执行一些脏（dirty）操作（`dirty_read`，`dirty_write`，`...`）。它们是在事务环境之外执行的操作。这些操作非常危险，只有在确定你的程序是单线程的，或者处于其他特殊情形时才可以使用它们。使用脏操作是出于效率的原因。

- SNMP表

  Mnesia有一个内建的SNMP表类型。它让实现SNMP管理系统变得非常简单。

Mnesia用户指南是最权威的Mnesia参考资料，它可以在Erlang分发套装的主站里找到。除此之外，Mnesia分发套装里的`examples`子目录（在我的机器上是`/usr/local/lib/erlang/lib/mnesia-X.Y.Z/examples`）里还有一些Mnesia范例。

现在我们已经完成了关于数据存储的两章。在下一章里，我们将来看看调试、跟踪和性能优化的方法。

20.9 练习

(1) 假设你要制作一个网站，让用户可以提供优秀Erlang程序的消息。制作一个包含三个表（`users`、`tips`和`abuse`）的Mnesia数据库来保存网站需要的所有数据。`users`表应当保存用户账户数据（姓名、邮箱地址和密码等）。`tips`表应当保存关于实用网站的消息（比如网站URL、描述和检查日期等）。`abuse`表应当保存某些数据来尽量防止网站滥用（比如网站访客的IP地址和网站访问次数等数据）。

配置这个数据库，让它运行在一台机器上并各有一个内存和磁盘副本。编写一些函数来读取、写入和列出这些表。

(2) 继续上一个练习，但要把数据库配置成在两台机器上各有内存和磁盘副本。尝试在一台机器上生成更新并让它崩溃，然后检查是否能接着访问第二台机器上的数据库。

(3) 编写一个查询来拒绝某个消息，条件是用户在一天内提交了超过10条消息，或者最近一天从三个以上的IP地址登录。

测量查询数据库所需的时间。

20

第 21 章

性能分析、调试与跟踪

在这一章里，我们将介绍一些技巧，你可以用它们来调整程序，寻找bug并避免错误。

- 性能分析（profiling）

  用来找出程序里有哪些热点区域，以便进行性能优化。我认为光凭猜测来找出程序的瓶颈几乎是不可能的。最好的做法是先编写我们的程序，然后确保它们是正确的，最后进行测量来找出时间都花在了哪里。当然，如果程序足够快，就可以省去最后这一步。

- 覆盖分析（coverage analysis）

  用来统计程序里的各行代码分别被执行了多少次。代码行的执行次数为零可能意味着存在错误或者是可以移除的无用代码。找出频繁执行的代码行也许能帮助你优化程序。

- 交叉引用（cross-referencing）

  用来找出是否有遗漏代码或者谁调用了什么。如果试图调用一个不存在的函数，交叉引用分析就会检测出来。这对那些拥有几十个模块的大型程序最为有用。

- 编译器诊断信息

  这一节将对编译器的诊断信息进行解释。

- 运行时错误消息

  运行时系统会生成许多不同的错误消息。我们将解释那些最常见的错误消息的含义。

- 调试方法

  调试内容分为两部分。首先会了解一些简单的技巧，然后来看Erlang调试器。

- 跟踪

  我们可以用进程跟踪来检查系统在运行过程中的行为。Erlang有一些高级跟踪工具，可以用来远程观察任意进程的行为。还能观察进程间的消息传递，找出进程内正在执行的函数，观察进程的调度情况，等等。

- 测试框架

  有许多测试框架可以进行Erlang程序的自动化测试。我们将简要概述这些框架，并提供它们的获取方式。

21.1 Erlang 代码的性能分析工具

标准Erlang分发套装自带了三种性能分析工具。

- cprof统计各个函数被调用的次数。它是一个轻量级的性能分析器，在活动系统上运行它会增加5% ~ 10%的系统负载。
- fprof显示调用和被调用函数的时间，结果会输出到一个文件。它适用于实验室或模拟系统里的大型系统性能分析，并会显著增加系统负载。
- eprof测量Erlang程序是如何使用时间的。它是fprof的前身，适用于小规模的性能分析。

下面演示如何运行cprof。我们将用它对17.6节里编写的代码进行性能分析：

```
1> cprof:start().          %% 启动性能分析器
4501
2> shout:start().          %% 运行应用程序
<0.35.0>
3> cprof:pause().          %% 暂停性能分析器
4844
4> cprof:analyse(shout). %% 分析函数调用
{shout,232,
[{{shout,split,2},73},
{{shout,write_data,4},33},
{{shout,the_header,1},33},
{{shout,send_file,6},33},
{{shout,bump,1},32},
{{shout,make_header1,1},5},
{{shout,'-got_request_from_client/3-fun-0-',1},4},
{{shout,songs_loop,1},2},
{{shout,par_connect,2},2},
{{shout,unpack_song_descriptor,1},1},
...
5> cprof:stop().           %% 停止性能分析器
4865
```

另外，cprof:analyse()会分析所有已收集统计数据的模块。

cprof的更多细节可以在网上找到[①]。

fprof[②]和eprof[③]大致类似于cprof。要了解更多细节，请查阅在线文档。

21.2 测试代码覆盖

我们在进行代码测试时，不光希望了解哪些代码行被频繁执行，还希望了解哪些代码行从未被执行。从未执行的代码行是潜在的错误来源，因此能找到它们的位置会很有帮助。要做到这一点，可以使用程序覆盖分析器。

① http://www.erlang.org/doc/man/cprof.html

② http://www.erlang.org/doc/man/fprof.html

③ http://www.erlang.org/doc/man/eprof.html

这里有一个例子：

```
1> cover:start().          %% 启动覆盖分析器
{ok,<0.34.0>}
2> cover:compile(shout).  %% 编译shout.erl来进行覆盖分析
{ok,shout}
3> shout:start().          %% 运行程序
<0.41.0>
Playing:<<"title: track018 performer: .. ">>
4> %% 让程序运行一段时间
4>  cover:analyse_to_file(shout).  %% 分析结果
{ok,"shout.COVER.out"}             %% 这是结果文件
```

它的结果会写入一个文件。

```
...
|  send_file(S, Header, OffSet, Stop, Socket, SoFar) ->
|      %% OffSet = 要播放的第一个字节
|      %% Stop   = 可以播放的最后一个字节
131..|      Need = ?CHUNKSIZE - byte_size(SoFar),
131..|      Last = OffSet + Need,
131..|      if
|         Last >= Stop ->
|             %% 数据不足，因此尽量续取并返回
0..|          Max = Stop - OffSet,
0..|          {ok, Bin} = file:pread(S, OffSet, Max),
0..|          list_to_binary([SoFar, Bin]);
|         true ->
131..|         {ok, Bin} = file:pread(S, OffSet, Need),
131..|         write_data(Socket, SoFar, Bin, Header),
131..|         send_file(S, bump(Header),
|                       OffSet + Need,  Stop, Socket, <<>>)
|      end.
...
```

在文件的左侧可以看到各条语句被执行的次数。用零标注的那些行值得特别注意。因为这些代码未被执行，所以我们无法确定这个程序是否正确。

最佳测试方法

对我们的代码执行覆盖分析能回答一个问题：哪些代码行从未被执行过？一旦知道哪些代码行未被执行，就可以设计测试案例来强制执行这些代码行。

要找出程序里出乎意料和隐藏较深的bug，这是一个万无一失的做法。每一行从未被执行的代码都有可能包含错误。强制执行这些代码行是我所知道的最佳程序测试方法。

我对最初的Erlang JAM[①]编译器就是这么做的。我记得在两年里只收到了三个bug报告，在此之后就没有bug报告了。

① JAM是Joe's Abstract Machine（Joe的抽象机）的缩写，它是首个Erlang编译器。

通过设计测试案例来让所有的覆盖统计数字大于零是一种很有效的做法，能够系统性地找出程序里的隐藏缺陷。

21.3 生成交叉引用

开发程序时，可以偶尔对代码运行一次交叉引用检查。如果有函数缺失，在程序运行之前就能发现它。

可以用xref模块来生成交叉引用。只有当代码在编译时设置了debug_info标识，xref才能正常工作。

我无法向你展示本书所附代码的xref输出，因为它们已经开发完成，不存在任何缺失的函数。不过，我会展示对我的某个业余项目代码运行交叉引用检查会发生什么。

vsg是一个简单的图形程序，我可能会在未来的某一天发布它。我们将对vsg目录里的代码进行分析，这是我用来进行程序开发的目录。

```
$ cd /home/joe/2007/vsg-1.6
$ rm *.beam
$ erlc +debug_info *.erl
$ erl
1> xref:d('.').
[{deprecated,[]},
{undefined,[{{new,win1,0},{wish_manager,on_destroy,2}},
{{vsg,alpha_tag,0},{wish_manager,new_index,0}},
{{vsg,call,1},{wish,cmd,1}},
{{vsg,cast,1},{wish,cast,1}},
{{vsg,mkWindow,7},{wish,start,0}},
{{vsg,new_tag,0},{wish_manager,new_index,0}},
{{vsg,new_win_name,0},{wish_manager,new_index,0}},
{{vsg,on_click,2},{wish_manager,bind_event,2}},
{{vsg,on_move,2},{wish_manager,bind_event,2}},
{{vsg,on_move,2},{wish_manager,bind_tag,2}},
{{vsg,on_move,2},{wish_manager,new_index,0}}]},
{unused,[{vsg,new_tag,0},
{vsg_indicator_box,theValue,1},
{vsg_indicator_box,theValue,1}]}]
```

xref:d('.')对当前目录里所有在编译时设置了调试标识的代码执行了一次交叉引用检查。它生成的列表包含了废弃（deprecated）、未定义（undefined）和未使用（unused）的函数。

像大多数工具一样，xref有一大堆选项，所以如果你想充分利用这个程序的强大功能，就必须去阅读手册。

21.4 编译器诊断信息

编译某个程序时，如果源代码存在语法错误，编译器就会向我们提供有用的错误消息。这些

消息大多数都是不言自明的：如果漏掉了括号、逗号或者关键字，编译器就会给出一个错误消息，内含问题语句所在的文件名和行号。下面是一些我们可能会遇到的错误。

21.4.1　头部不匹配

如果函数定义里的各个子句不具有相同的名称和元数，就会得到以下错误：

bad.erl

```
foo(1,2) ->
    a;
foo(2,3,a) ->
    b.
```

```
1> c(bad).
./bad.erl:3: head mismatch
```

21.4.2　未绑定变量

这里有一些包含未绑定变量的代码：

bad.erl

```
Line 1 foo(A, B) ->
     2     bar(A, dothis(X), B),
     3     baz(Y, X).
```

```
1> c(bad).
./bad.erl:2: variable 'X' is unbound
./bad.erl:3: variable 'Y' is unbound
```

它的意思是第2行里的变量X没有值。这个错误实际上不在第2行，却是在第2行里检测到的，因为那里是未绑定变量X首次出现的位置。（第3行也用到了X，但是编译器只会报告错误首次出现的行。）

21.4.3　未结束字符串

如果我们忘了给字符串或原子加上后引号，就会得到以下错误消息：

unterminated string starting with "..."

有时候，找到缺失的引号可能很困难。如果你看到这个消息但就是找不到哪里缺了引号，不妨把一个引号放在你认为可能是问题所在的位置附近，然后编译程序。这样也许就能生成更详细的诊断信息，帮助你准确定位错误。

21.4.4　不安全变量

如果我们编译下列代码：

bad.erl

```
foo() ->
    case bar() of
        1 ->
            X = 1,
            Y = 2;
        2 ->
            X = 3
    end,
    b(X).
```

就会得到以下警告：

```
1> c(bad).
./bad.erl:5: Warning: variable 'Y' is unused
{ok,bad}
```

这仅仅是一个警告，因为代码里定义了Y却没有使用它。如果现在把程序修改如下：

bad.erl

```
foo() ->
    case bar() of
        1 ->
            X = 1,
            Y = 2;
        2 ->
            X = 3
    end,
    b(X, Y).
```


就会得到以下错误：

```
> c(bad).
./bad.erl:9: variable 'Y' unsafe in 'case' (line 2)
{ok,bad}
```

编译器推断出程序可能会进入case表达式的第二个分支（这时变量Y是未定义的），因此它生成了一个“不安全变量”的错误消息。

21.4.5 影子变量

影子变量（shadowed variable）的意思是某些变量屏蔽了之前定义的其他变量的值，从而使你无法使用那些值。这里有一个例子：

bad.erl

```
foo(X, L) ->
    lists:map(fun(X) -> 2*X end, L).
```

```
1> c(bad).
./bad.erl:1: Warning: variable 'X' is unused
./bad.erl:2: Warning: variable 'X' shadowed in 'fun'
{ok,bad}
```

编译器担心我们可能在程序里犯了个错误。我们在fun里计算2*X，但指的是哪个X：是作为fun参数的X还是作为foo参数的X?

如果发生了这种情况，最好的做法是重命名其中某一个X，从而避免警告。可以重新编写代码如下：

bad.erl

```
foo(X, L) ->
    lists:map(fun(Z) -> 2*Z end, L).
```

现在在fun定义里使用X就不会有问题了。

21.5　运行时诊断

如果某个Erlang进程崩溃了，我们可能会得到一个错误消息。要看到这个消息，就必须有别的进程监视这个崩溃进程，并在它崩溃时打印出错误消息。如果只是用spawn创建了一个进程，那么当它崩溃时就不会得到任何错误消息。要想看到所有错误消息，最好的方法就是始终使用spawn_link。

栈跟踪

每当一个与shell相连的进程崩溃后，shell就会打印出栈跟踪信息。为了看到这些信息，我们将特意编写一个简单的错误函数，然后在shell里调用它。

lib_misc.erl

```
deliberate_error(A) ->
    bad_function(A, 12),
    lists:reverse(A).

bad_function(A, _) ->
    {ok, Bin} = file:open({abc,123}, A),
    binary_to_list(Bin).
```

```
1> lib_misc:deliberate_error("file.erl").
** exception error: no match of right hand side value {error,badarg}
in function  lib_misc:bad_function/2 (lib_misc.erl, line 804)
in call from lib_misc:deliberate_error/1 (lib_misc.erl, line 800)
```

当调用lib_misc:deliberate_error("file.erl")时会产生一个错误,随后系统会打印出错误消息和栈跟踪信息。这个错误消息如下：

```
** exception error: no match of right hand side value {error,badarg}
```

它来源于下面这一行：

```
{ok, Bin} = file:open({abc,123}, A)
```

调用`file:open/2`返回了`{error, badarg}`，这是因为`{abc,123}`不是一个合法的`file:open`输入值。当我们试图用`{ok, Bin}`匹配返回值时，就会得到一个不匹配错误，运行时系统则打印出了`** exception error ... {error, badarg}`。

错误消息之后是栈跟踪信息。它以发生错误的函数名开头，后面是当前函数完成后将会返回的各个函数清单（包括函数名、模块名和行号）。由此可知，错误发生在`lib_misc:bad_function/2`里，而此函数将会返回到`lib_misc:deliberate_error/1`，以此类推。

请注意，只有栈跟踪信息顶部的那些条目才真正值得注意。如果出错函数的调用序列包含一个尾调用，那它是不会出现在栈跟踪信息里的。举个例子，如果定义`deliberate_error1`函数如下：

lib_misc.erl

```
deliberate_error1(A) ->
    bad_function(A, 12).
```


那么当我们调用`deliberate_error1`函数并得到错误时，栈跟踪信息里将不会出现这个函数。

```
2> lib_misc:deliberate_error1("file.erl").
** exception error: no match of right hand side value {error,badarg}
in function  lib_misc:bad_function/2 (lib_misc.erl, line 804)
```

跟踪信息里没有`deliberate_error1`调用是因为`bad_function`是在`deliberate_error1`的最后调用的，它完成后不会返回到`deliberate_error1`，而是返回到`deliberate_error1`的调用者。

（这是*尾调用*优化的结果。如果某个函数里最后执行的表达式是一个函数调用，那么实际上会被替换为跳跃。如果没有这种优化，在消息接收循环代码里使用的*无限循环*编程模式就无法实现。然而，正是因为有了这种优化，调用函数实际上会在调用栈里替换成被调用函数，因此在栈跟踪信息里不可见。）

21.6 调试方法

调试Erlang代码非常简单。你可能会很惊讶，但这是一次性赋值变量的功劳。因为Erlang没有指针或可变状态（ETS表和进程字典除外），所以通常很容易找到问题的所在。一旦观察到某个变量的值不正确，就能相对容易地找到发生的时间和地点。

我发现调试器在编写C程序时是一种非常有用的工具，因为我可以让它监视变量并告诉我变量值何时发生变化。在很多时候这一点很重要，因为C里的内存可以通过指针间接修改。你可能

很难知道对某一个内存块的修改来自何处。我不觉得Erlang有相同的调试器需求，因为我们不能通过指针修改状态。

Erlang程序员使用各种各样的方法来调试他们的程序。到目前为止，最常用的方法就是给有问题的程序添加打印语句。但如果想查看的数据结构变得非常大，这种方法就无效了，在这种情况下可以把它们转储到一个文件，留待将来检查。

一些人使用错误记录器来保存错误消息，另一些人则把它们写入文件。如果都实现不了，还可以使用Erlang调试器或者跟踪程序的执行过程。让我们来看看这几种方法。

21.6.1　io:format调试

给程序添加打印语句是最常见的调试形式。可以简单地在程序的关键位置添加io:format(...)语句来打印出感兴趣的变量值。

调试并行程序时，一种好的做法是在发送消息到别的进程之前先把它打印出来，收到消息之后也要立即打印。

我在编写并发程序时，通常会这样开始编写接收循环：

```
loop(...) ->
    receive
        Any ->
           io:format("*** warning unexpected message:~p~n",[Any])
           loop(...)
    end.
```

随后，在给接收循环添加模式的过程中，如果进程收到任何无法理解的消息，我就会看到打印出的警告消息。我还会用spawn_link代替spawn，从而确保一旦进程异常退出，错误消息就会被打印出来。

我经常使用一个宏：NYI（“not yet implemented”的缩写，意思是尚未实现）。它的定义如下：

lib_misc.erl

```
-define(NYI(X),(begin
                  io:format("*** NYI ~p ~p ~p~n",[?MODULE, ?LINE, X]),
                  exit(nyi)
              end)).
```

然后我就可以像这样使用这个宏：

lib_misc.erl

```
glurk(X, Y) ->
    ?NYI({glurk, X, Y}).
```

函数glurk的主体尚未编写，所以当我调用glurk时，程序就会崩溃。

```
> lib_misc:glurk(1,2).
*** NYI lib_misc 83 {glurk,1,2}
** exited: nyi *
```

程序崩溃后显示了一个错误消息，这样我就知道是时候完整实现这个函数了。

21.6.2 转储至文件

如果感兴趣的数据结构很大，就可以把它写入一个文件，做法是使用像dump/2这样的函数。

lib_misc.erl

```
dump(File, Term) ->
    Out = File ++ ".tmp",
    io:format("** dumping to ~s~n",[Out]),
    {ok, S} = file:open(Out, [write]),
    io:format(S, "~p.~n",[Term]),
    file:close(S).
```

它会打印一个警告消息来提醒我们有新文件被创建，并给文件添加一个.tmp扩展名（这样以后就能轻松删除所有的临时文件）。然后它会把我们感兴趣的数据类型美化打印到一个文件里。可以在以后的某个时间用文本编辑器检查该文件。这个方法很简单，而且特别适合检查大型的数据结构。

21.6.3 使用错误记录器

可以用错误记录器来创建一个保存调试输出的文本文件。要做到这一点，我们将创建一个像下面这样的配置文件：

elog5.config

```
%% 文本错误日志
[ {kernel,
  [{error_logger,
    {file, "/Users/joe/error_logs/debug.log"}}]}].
```

然后用下列命令启动Erlang：

```
erl -config elog5.config
```

配置文件里指定的文件会保存所有通过调用error_logger模块里的函数所创建的错误消息，以及所有在shell里打印出的错误消息。

21.7 Erlang 调试器

标准的Erlang分发套装包含了一个调试器。我不会在这里过多介绍它，只会告诉你如何启动它并列出一些文档地址。启动之后，调试器的用法很简单。你可以检查变量、单步执行代码、设置断点或进行其他操作。

因为我们经常想调试多个进程，所以调试器自身能分裂出许多副本，这样我们就能看到多个调试窗口，每个窗口对应一个正在调试的进程。

唯一的难点是启动这个调试器。

```
1> %% 重新编译lib_misc，这样就能调试它了
1>  c(lib_misc, [debug_info]).
{ok, lib_misc}
2> im().  %% 这里会弹出一个窗口，现在可以忽略它
<0.42.0>
3> ii(lib_misc).
{module,lib_misc}
4> iaa([init]).
true.
5> lib_misc:
...
```

运行这些命令会打开图21-1所示的窗口。

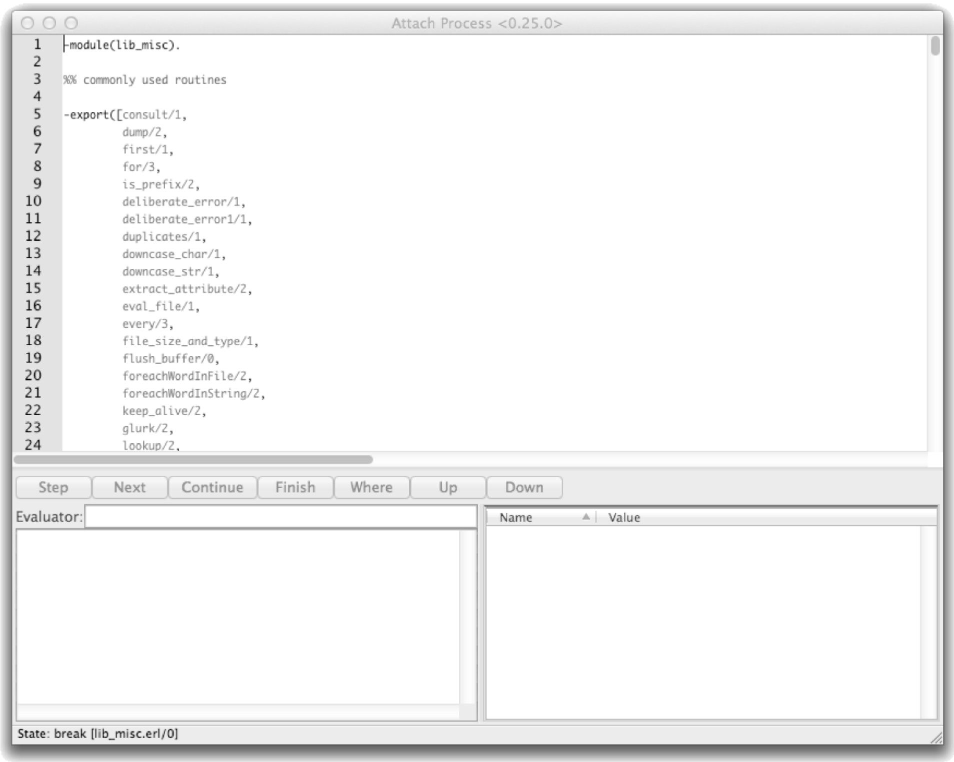


图21-1　调试器初始窗口

可以在调试器里设置断点、检查变量或进行其他操作。

所有不带模块前缀（`ii/1`和`iaa/1`等）的命令都是从模块`i`里导出的。它是调试器/解释器接口模块。这些方法可以在shell里直接访问，无需添加模块前缀。

我们在启动调试器的过程中调用了一些函数，它们执行以下任务。

- im()
  启动一个新的图形监视器，它是调试器的主窗口，会显示调试器正在监视的所有进程的状态。
- ii(Mod)
  解释Mod模块里的代码。
- iaa([init])
  在执行已解释代码的进程启动时让调试器关联它。

要了解更多关于调试的信息，可以试试这些资源。

- http://www.erlang.org/doc/apps/debugger/debugger.pdf
  这份调试器参考手册对调试器进行了介绍，包括屏幕截图和API文档，等等。它是调试器高级用户的必读文档。
- http://www.erlang.org/doc/man/i.html
  在这里能找到shell里可用的调试器命令。

21.8　跟踪消息与进程执行

无需用特殊方式编译你的代码就能跟踪某个进程。跟踪一个（或多个）进程是一种强有力的方式，它既能让你理解系统行为，又能在不改动代码的前提下测试复杂系统。它特别适用于嵌入式系统，或者在无法修改被测代码的时候使用。

可以在底层调用一些Erlang内置函数来设置一个跟踪。用这些内置函数设置复杂的跟踪很困难，所以我们设计了一些库来让这个任务变得容易些。

我们将从底层的Erlang内置跟踪函数入手，看看如何设置一个简单的跟踪器，然后展示能为内置跟踪函数提供更高层接口的库。

对底层跟踪而言，有两个内置函数特别重要。erlang:trace/3的作用大致是，“我想要监视这个进程，所以请在发生有意思的事时给我发一个消息”。erlang:trace_pattern则定义了哪些算是“有意思”的事。

- erlang:trace(PidSpec, How, FlagList)
  它会启动跟踪。PidSpec告诉系统要跟踪什么进程，How是一个开启或关闭跟踪的布尔值，FlagList指定了要跟踪的事件（比如，可以跟踪所有的函数调用，跟踪所有正在发送的消息，跟踪垃圾收集何时进行，等等）。
  一旦调用了erlang:trace/3这个内置函数，调用它的进程就会在跟踪事件发生时收到跟踪消息。跟踪事件本身是通过调用erlang:trace_pattern/3确定的。
- erlang:trace_pattern(MFA, MatchSpec, FlagList)
  它用于设置一个*跟踪模式*。如果模式匹配，请求的操作就会执行。这里的MFA是一个{Module, Function, Args}元组，指定要对哪些代码应用跟踪模式。MatchSpec是一个模式，会在每次进入MFA指定的函数时进行测试，而FlagList规定了跟踪条件满足时要做什么。

为MatchSpec编写匹配规则非常复杂，对我们理解跟踪也没有太大帮助。好在有几个库[1]能够让事情变得简单一些。

可以用之前的两个内置函数编写一个简单的跟踪器。trace_module(Mod, Fun)会对Mod模块设置跟踪，然后执行Fun()。我们想要跟踪Mod模块里的所有函数调用和返回值。

tracer_test.erl

```
trace_module(Mod, StartFun) ->
    %% 分裂一个进程来执行跟踪
    spawn(fun() -> trace_module1(Mod, StartFun) end).

trace_module1(Mod, StartFun) ->
    %% 下一行的意思是：跟踪Mod里的
    %% 所有函数调用和返回值
    erlang:trace_pattern({Mod, '_','_'},
                         [{'_',[],[{return_trace}]}],
                         [local]),
    %% 分裂一个函数来执行跟踪
    S = self(),
    Pid = spawn(fun() -> do_trace(S, StartFun) end),
    %% 设置跟踪，告诉系统开始
    %% 跟踪进程Pid
    erlang:trace(Pid, true, [call,procs]),
    %% 现在让Pid启动
    Pid ! {self(), start},
    trace_loop().

%% do_trace会在Parent的指示下
%% 执行StartFun()
do_trace(Parent, StartFun) ->
    receive
        {Parent, start} ->
            StartFun()
    end.

%% trace_loop负责显示函数调用和返回值
trace_loop() ->
    receive
        {trace,_,call, X} ->
            io:format("Call: ~p~n",[X]),
            trace_loop();
        {trace,_,return_from, Call, Ret} ->
            io:format("Return From: ~p => ~p~n",[Call, Ret]),
            trace_loop();
        Other ->
```

[1] http://www.erlang.org/doc/man/ms_transform.html

```
            %% 我们得到了其他的一些消息，打印它们
            io:format("Other = ~p~n",[Other]),
            trace_loop()
    end.
```

现在定义一个测试案例如下：

tracer_test.erl

```
test2() ->
    trace_module(tracer_test, fun() -> fib(4) end).

fib(0) -> 1;
fib(1) -> 1;
fib(N) -> fib(N-1) + fib(N-2).
```

然后就能跟踪我们的代码了。

```
1> c(tracer_test).
{ok,tracer_test}
2> tracer_test:test2().
<0.42.0>Call: {tracer_test,'-trace_module1/2-fun-0-',
[<0.42.0>,#Fun<tracer_test.0.36786085>]}
Call: {tracer_test,do_trace,[<0.42.0>,#Fun<tracer_test.0.36786085>]}
Call: {tracer_test,'-test2/0-fun-0-',[]}
Call: {tracer_test,fib,[4]}
Call: {tracer_test,fib,[3]}
Call: {tracer_test,fib,[2]}
Call: {tracer_test,fib,[1]}
Return From: {tracer_test,fib,1} => 1
Call: {tracer_test,fib,[0]}
Return From: {tracer_test,fib,1} => 1
Return From: {tracer_test,fib,1} => 2
Call: {tracer_test,fib,[1]}
Return From: {tracer_test,fib,1} => 1
Return From: {tracer_test,fib,1} => 3
Call: {tracer_test,fib,[2]}
Call: {tracer_test,fib,[1]}
Return From: {tracer_test,fib,1} => 1
Call: {tracer_test,fib,[0]}
Return From: {tracer_test,fib,1} => 1
Return From: {tracer_test,fib,1} => 2
Return From: {tracer_test,fib,1} => 5
Return From: {tracer_test,'-test2/0-fun-0-',0} => 5
Return From: {tracer_test,do_trace,2} => 5
Return From: {tracer_test,'-trace_module1/2-fun-0-',2} => 5
Other = {trace,<0.43.0>,exit,normal}
```

跟踪器的输出可能会极其详细，对理解程序的动态行为很有价值。阅读代码能带给我们静态的系统印象，但观察消息流能让我们看到系统动态行为的景象。

21

使用跟踪库

可以用库模块dbg来执行与之前相同的跟踪。它会隐藏底层Erlang内置函数的所有细节。

tracer_test.erl

```
test1() ->
    dbg:tracer(),
    dbg:tpl(tracer_test,fib,'_',
            dbg:fun2ms(fun(_) -> return_trace() end)),
    dbg:p(all,[c]),
    tracer_test:fib(4).
```

运行它会得到以下输出：

```
1> tracer_test:test1().
(<0.34.0>) call tracer_test:fib(4)
(<0.34.0>) call tracer_test:fib(3)
(<0.34.0>) call tracer_test:fib(2)
(<0.34.0>) call tracer_test:fib(1)
(<0.34.0>) returned from tracer_test:fib/1 -> 1
(<0.34.0>) call tracer_test:fib(0)
(<0.34.0>) returned from tracer_test:fib/1 -> 1
(<0.34.0>) returned from tracer_test:fib/1 -> 2
(<0.34.0>) call tracer_test:fib(1)
(<0.34.0>) returned from tracer_test:fib/1 -> 1
(<0.34.0>) returned from tracer_test:fib/1 -> 3
(<0.34.0>) call tracer_test:fib(2)
(<0.34.0>) call tracer_test:fib(1)
(<0.34.0>) returned from tracer_test:fib/1 -> 1
(<0.34.0>) call tracer_test:fib(0)
(<0.34.0>) returned from tracer_test:fib/1 -> 1
(<0.34.0>) returned from tracer_test:fib/1 -> 2
(<0.34.0>) returned from tracer_test:fib/1 -> 5
```

这里用库代码而不是内置跟踪函数实现了前一节的操作。要实现细粒度的控制，多半应该用内置跟踪函数来编写自己的自定义跟踪代码。要进行快速试验，库代码就够用了。

要了解更多有关跟踪的信息，你需要阅读以下三个模块的手册页。

- dbg提供了Erlang内置跟踪函数的简化接口。
- ttb是内置跟踪函数的另一种接口，比dbg更高层。
- ms_transform能生成用于跟踪软件的匹配规则。

21.9　Erlang 代码的测试框架

应该为复杂的项目设立一个测试框架，并把它集成到你的构建系统里。如果有兴趣，可以探索一下下面这两个框架。

- 通用测试框架

  通用测试框架（Common Test Framework）是Erlang/OTP分发套装的一部分。它提供了一套完整的工具来进行自动化测试。通用测试框架被用来测试Erlang分发套装本身以及爱立信公司的许多产品。

- 基于属性的测试

  基于属性的测试相对比较新，它是一种极其优秀的技巧，能够抖落出代码里难以发现的bug。我们不需要编写测试案例，而是用一种谓词逻辑（predicate logic）形式来描述系统的各个属性。测试工具会生成符合系统属性的随机测试案例，然后检查这些属性是否会出现违规。

  有两个基于属性的工具可以用来测试Erlang程序：一个是QuickCheck，它是一家名为Quviq[①]的瑞典公司推出的商业性程序；另一个是proper[②]，它的灵感来源于QuickCheck。

祝贺你！现在你已经了解了如何编写顺序和并发程序，了解了文件和套接字、数据存储和数据库，以及如何调试和测试程序。

在下一章里，我们将换一个话题，来讨论开放电信平台（Open Telecom Platform，简称OTP）。这个古怪的名字反映了Erlang的历史。OTP是一个应用程序框架，或者说一组编程模式，它能简化分布式容错系统的编码工作。这个久经考验的框架已被大量应用程序使用，所以把它用于你自己的项目是一个很好的起点。

21.10 练习

(1) 创建一个新目录，然后复制标准库模块`dict.erl`到这个目录里。给`dict.erl`添加一个错误，使它在某一行代码被执行时会崩溃。然后编译这个模块。

(2) 现在我们有了一个问题模块`dict`，但多半还不知道它有问题，所以需要引发错误。编写一个简单的测试模块来用多种方式调用`dict`，看看能否让`dict`崩溃。

(3) 使用覆盖分析器来检查`dict`里的每一行代码各被执行了多少次。给你的测试模块添加更多的测试案例，看看是否覆盖了`dict`里的所有代码。这么做的目的是确保`dict`里的每一行代码都被执行。一旦知道哪些代码行未被执行，就能轻松进行反向推导，找出测试案例里的哪些代码行能导致某一行原代码被执行。

坚持做这件事，直到程序崩溃为止。崩溃迟早会发生，因为当每一行代码都被覆盖时，就意味着错误已被触发。

(4) 现在我们有了一个错误。假装你不知道错误出在哪里，然后使用这一章里介绍的方法来找出这个错误。

当你真的不知道错误在哪里时，这个练习会更有效。找个朋友来破坏你的某些模块，然后对这些模块运行覆盖测试来引发错误。一旦引发了错误，就使用调试技术来找出问题所在。

---

① http://www.quviq.com

② https://github.com/manopapad/proper

第 22 章

OTP介绍

OTP代表Open Telecom Platform（开放电信平台）。这个名字其实有一些误导性，因为OTP比你想象的要通用得多。它是一个应用程序操作系统，包含了一组库和实现方式，可以构建大规模、容错和分布式的应用程序。它由瑞典电信公司爱立信开发，在爱立信内部用于构建容错式系统。标准的Erlang分发套装包含OTP库。

OTP包含了许多强大的工具，例如一个完整的Web服务器，一个FTP服务器和一个CORBA ORB[①]等，它们全都是用Erlang编写的。OTP还包含了构建电信应用程序的最先进工具，能够实现H248、SNMP和ASN.1/Erlang交叉编译器（这些是电信行业里常用的协议）。我不会在这里讨论它们，你可以在Erlang网站[②]上找到和这些主题相关的大量信息。

如果想用OTP编写自己的应用程序，你会发现一个很有用的核心概念是OTP行为。行为封装了常见的行为模式，你可以把它看作是一个用回调函数作为参数的应用程序框架。

OTP的威力来自于行为本身就能提供容错性、可扩展性和动态代码升级等属性。换句话说，回调函数的编写者不必担心容错之类的事情，因为行为已经提供了它们。熟悉Java的读者可以把行为看作是一个J2EE容器。

简单地说，行为负责解决问题的非函数部分，而回调函数负责解决函数部分。这么做的优点在于问题的非函数部分（比如如何进行实时代码升级）对所有应用程序都是一样的，而函数部分（由回调函数提供）在每个问题里都是不同的。

在这一章里，我们将非常详细地介绍其中一种行为：`gen_server`模块。但是，在深入`gen_server`工作方式的核心细节之前，首先会从一个简单的服务器（我们能想象的最简单的服务器）入手，然后一步步改进它，直到实现`gen_server`模块的完整功能。这样你就能切实理解`gen_server`是如何工作的，并为深入探索做好准备。

本章的规划如下。

(1) 用Erlang编写一个客户端-服务器小程序。

(2) 慢慢让这个程序通用化，并添加一些特性。

(3) 转向真正的代码。

① http://en.wikipedia.org/wiki/Common_Object_Request_Broker_Architecture

② http://www.erlang.org/

22.1　通用服务器之路

这是整本书里最重要的一节，所以请读一遍，再读两遍，甚至读一百遍，确保你能完全理解里面的内容。

这一节是关于构建抽象的，我们将看到一个名为gen_server.erl的服务器。gen server（通用服务器）是OTP系统里最常用的抽象，但很多人从来没有深入探寻过gen_server.erl是如何工作的。一旦理解了gen server是如何构建的，就能重复这个抽象过程来构建自己的抽象。

我们将编写四个小小的服务器：server1、server2、server3和server4，每一个服务器都与上一个稍有不同。server4会类似于Erlang分发套装里的gen server。我们的目标是把问题的非函数部分与函数部分完全分开。这句话现在也许对你没有什么意义，但是请别担心，很快就会有了。深呼吸！

22.1.1　Server 1：基本的服务器

以下代码是我们的首次尝试。它是一个小小的服务器，可以用回调模块作为它的参数。

server1.erl

```
-module(server1).
-export([start/2, rpc/2]).
start(Name, Mod) ->
    register(Name, spawn(fun() -> loop(Name, Mod, Mod:init()) end)).
rpc(Name, Request) ->
    Name ! {self(), Request},
    receive
        {Name, Response} -> Response
    end.
loop(Name, Mod, State) ->
    receive
        {From, Request} ->
            {Response, State1} = Mod:handle(Request, State),
            From ! {Name, Response},
            loop(Name, Mod, State1)
    end.
```

这一小段代码凝聚了服务器的精华。下面我们给server1编写一个回调模块，它是一个名称服务器回调模块：

name_server.erl

```
-module(name_server).
-export([init/0, add/2, find/1, handle/2]).
-import(server1, [rpc/2]).

%% 客户端方法
```

22

```
add(Name, Place) -> rpc(name_server, {add, Name, Place}).
find(Name)       -> rpc(name_server, {find, Name}).

%% 回调方法
init() -> dict:new().
handle({add, Name, Place}, Dict) -> {ok, dict:store(Name, Place, Dict)};
handle({find, Name}, Dict)       -> {dict:find(Name, Dict), Dict}.
```

这段代码实际上执行两个任务。它首先充当被服务器框架代码调用的回调模块，与此同时，它还包含了将被客户端调用的接口方法。OTP的惯例是把这两类函数放在同一个模块里。

为了证明它能工作，你可以这么做：

```
1> server1:start(name_server, name_server).
true
2> name_server:add(joe, "at home").
ok
3> name_server:find(joe).
{ok,"at home"}
```

现在停下来想一想。这个回调模块没有用于并发的代码，没有分裂，没有发送，没有接收，也没有注册。它是纯粹的顺序代码，别无其他。这就意味着我们可以在完全不了解底层并发模型的情况下编写客户端–服务器模型。

这就是所有服务器的基本模式。一旦理解了基本的结构，就可以轻轻松松地“自主研发”了。

22.1.2　Server 2：实现事务的服务器

这个服务器会在查询产生异常错误时让客户端崩溃：

server2.erl

```
-module(server2).
-export([start/2, rpc/2]).

start(Name, Mod) ->
    register(Name, spawn(fun() -> loop(Name,Mod,Mod:init()) end)).

rpc(Name, Request) ->
    Name ! {self(), Request},
    receive
        {Name, crash} -> exit(rpc);
        {Name, ok, Response} -> Response
    end.

loop(Name, Mod, OldState) ->
    receive
        {From, Request} ->
            try Mod:handle(Request, OldState) of
                {Response, NewState} ->
```

```
                    From ! {Name, ok, Response},
                    loop(Name, Mod, NewState)
            catch
                _:Why ->
                    log_the_error(Name, Request, Why),
                    %% 发送一个消息来让客户端崩溃
                    From ! {Name, crash},
                    %% 以*初始*状态继续循环
                    loop(Name, Mod, OldState)
            end
    end.

log_the_error(Name, Request, Why) ->
    io:format("Server ~p request ~p ~n"
              "caused exception ~p~n",
              [Name, Request, Why]).
```

这段代码在服务器里实现了“事务语义”，它会在处理函数抛出异常错误时用State（状态）的初始值继续循环。但如果处理函数成功了，它就会用处理函数提供的NewState值继续循环。

当处理函数失败时，服务器会给发送问题消息的客户端发送一个消息，让它崩溃。这个客户端不能继续工作，因为它发送给服务器的请求导致了处理函数的崩溃，但其他想要使用服务器的客户端不会受到影响。另外，当处理函数发生错误时，服务器的状态不会改变。

请注意，这个服务器使用的回调模块和用于server1的回调模块一模一样。通过修改服务器并保持回调模块不变，我们就能修改回调模块的非函数行为。

22

注意 最后那句话并不完全正确。从server1转到server2时，我们必须对回调模块做一点小小的改动，也就是把-import声明里的server1改成server2。除此之外并无其他改动。

22.1.3 Server 3：实现热代码交换的服务器

现在我们将添加热代码交换（hot code swapping）功能。大多数服务器都执行一个固定的程序，如果要修改服务器的行为，就必须先停止服务器，再用修改后的代码重启它。而要修改这个服务器的行为，不用停止它，只需要发送一个包含新代码的消息，它就会提取新代码，然后用新代码和老的会话数据继续工作。这一过程被称为热代码交换。

server3.erl

```
-module(server3).
-export([start/2, rpc/2, swap_code/2]).
start(Name, Mod) ->
    register(Name,
             spawn(fun() -> loop(Name,Mod,Mod:init()) end)).
swap_code(Name, Mod) -> rpc(Name, {swap_code, Mod}).
rpc(Name, Request) ->
```

```
    Name ! {self(), Request},
    receive
        {Name, Response} -> Response
    end.
loop(Name, Mod, OldState) ->
    receive
        {From, {swap_code, NewCallBackMod}} ->
            From ! {Name, ack},
            loop(Name, NewCallBackMod, OldState);
        {From, Request} ->
            {Response, NewState} = Mod:handle(Request, OldState),
            From ! {Name, Response},
            loop(Name, Mod, NewState)
    end.
```

如果向服务器发送一个交换代码消息，它就会把回调模块改为消息里包含的新模块。

我们可以演示这一点，做法是用某个回调模块启动server3，然后动态交换这个回调模块。不能用name_server作为回调模块，因为服务器名已经被硬编译进这个模块里了。因此，将制作一个名为name_server1的副本，然后在里面修改服务器的名称。

name_server1.erl

```
-module(name_server1).
-export([init/0, add/2, find/1, handle/2]).
-import(server3, [rpc/2]).

%% 客户端方法
add(Name, Place) -> rpc(name_server, {add, Name, Place}).
find(Name)       -> rpc(name_server, {find, Name}).

%% 回调方法
init() -> dict:new().

handle({add, Name, Place}, Dict) -> {ok, dict:store(Name, Place, Dict)};
handle({find, Name}, Dict)       -> {dict:find(Name, Dict), Dict}.
```

首先将用回调模块name_server1启动server3。

```
1> server3:start(name_server, name_server1).
true
2> name_server1:add(joe, "at home").
ok
3> name_server1:add(helen, "at work").
ok
```

现在假设想要找出这个名称服务器能提供的所有名称。API里没有函数能做到这一点，因为name_server模块只包含访问函数add和find。

于是我们以闪电般的速度打开文本编辑器并编写一个新的回调模块。

new_name_server.erl

```
-module(new_name_server).
-export([init/0, add/2, all_names/0, delete/1, find/1, handle/2]).
-import(server3, [rpc/2]).

%% 接口
all_names()       -> rpc(name_server, allNames).
add(Name, Place) -> rpc(name_server, {add, Name, Place}).
delete(Name)      -> rpc(name_server, {delete, Name}).
find(Name)        -> rpc(name_server, {find, Name}).
%% 回调方法
init() -> dict:new().

handle({add, Name, Place}, Dict) -> {ok, dict:store(Name, Place, Dict)};
handle(allNames, Dict)           -> {dict:fetch_keys(Dict), Dict};
handle({delete, Name}, Dict)     -> {ok, dict:erase(Name, Dict)};
handle({find, Name}, Dict)       -> {dict:find(Name, Dict), Dict}.
```

编译这个模块并告知服务器交换它的回调模块。

```
4> c(new_name_server).
{ok,new_name_server}
5> server3:swap_code(name_server, new_name_server).
ack
```

现在就可以运行服务器里的新函数了。

```
6> new_name_server:all_names().
[joe,helen]
```

22

我们在这里实时更换了回调模块，这就是动态代码升级，就发生在你的眼前，没有什么黑魔法。

现在再停下来想一想。之前完成的两个任务通常都被认为很有难度，事实的确如此。编写能实现“事务语义”的服务器很困难，编写能实现动态代码升级的服务器也很困难，但这个方法让它们变得简单了。

这个方法极其强大。传统上我们认为服务器是有状态的程序，当我们向它发送消息时会改变它的状态。服务器里的代码在首次调用时就固定了，如果想要修改服务器里的代码，就必须停止服务器并修改代码，然后重启服务器。在前面的例子中，修改服务器的代码就像修改服务器的状态那样简单。我们用这个方法编写了许多产品，它们从来不会因为软件维护升级而停止服务。

22.1.4 Server 4：事务与热代码交换

在前两个服务器里，代码升级和事务语义是分开的。现在我要把它们组合到一个服务器里。好戏开始了。

server4.erl

```
-module(server4).
-export([start/2, rpc/2, swap_code/2]).

start(Name, Mod) ->
    register(Name, spawn(fun() -> loop(Name,Mod,Mod:init()) end)).
swap_code(Name, Mod) -> rpc(Name, {swap_code, Mod}).
rpc(Name, Request) ->
    Name ! {self(), Request},
    receive
        {Name, crash} -> exit(rpc);
        {Name, ok, Response} -> Response
    end.

loop(Name, Mod, OldState) ->
    receive
        {From, {swap_code, NewCallbackMod}} ->
            From ! {Name, ok, ack},
            loop(Name, NewCallbackMod, OldState);
        {From, Request} ->
            try Mod:handle(Request, OldState) of
                {Response, NewState} ->
                    From ! {Name, ok, Response},
                    loop(Name, Mod, NewState)
            catch
                _: Why ->
                    log_the_error(Name, Request, Why),
                    From ! {Name, crash},
                    loop(Name, Mod, OldState)
            end
    end.

log_the_error(Name, Request, Why) ->
    io:format("Server ~p request ~p ~n"
              "caused exception ~p~n",
              [Name, Request, Why]).
```

这个服务器同时提供了热代码交换和事务语义，干净利落！

22.1.5　Server 5：更多乐趣

理解动态代码变换的概念之后，我们就能找到更多乐趣。这里有一个服务器，它不会做任何事，直到你通知它变成某一种类型的服务器：

server5.erl

```
-module(server5).
-export([start/0, rpc/2]).
start() -> spawn(fun() -> wait() end).
```

```
wait() ->
    receive
        {become, F} -> F()
    end.
rpc(Pid, Q) ->
    Pid ! {self(), Q},
    receive
        {Pid, Reply} -> Reply
    end.
```

如果启动它并向它发送一个{become, F}消息，它就会变成一个执行F()函数的F服务器。先来启动它。

```
1> Pid = server5:start().
<0.57.0>
```

我们的服务器不做任何事，只是在等待一个become消息。

现在来定义一个服务器函数。没什么复杂的，只是计算阶乘而已。

my_fac_server.erl

```
-module(my_fac_server).
-export([loop/0]).

loop() ->
    receive
        {From, {fac, N}} ->
            From ! {self(), fac(N)},
            loop();
        {become, Something} ->
            Something()
    end.

fac(0) -> 1;
fac(N) -> N * fac(N-1).
```

确保它成功编译之后，就可以通知进程<0.57.0>变成一个阶乘服务器了。

```
2> c(my_fac_server).
{ok,my_fac_server}
3> Pid ! {become, fun my_fac_server:loop/0}.
{become,#Fun<my_fac_server.loop.0>}
```

现在这个进程已经变成一个阶乘服务器了，我们来调用它。

```
4> server5:rpc(Pid, {fac,30}).
265252859812191058636308480000000
```

这个进程会一直扮演阶乘服务器的角色，直到向它发送一个{become, Something}消息来告诉它做点别的什么。

正如你在前面这些例子中所看见的，我们可以制作各种不同类型的服务器，让它们具有不同

的语义和一些相当惊人的属性。这个方法实在是太过强大，如果彻底发挥它的潜力，就能生成拥有惊人威力和美感的小程序。如果我们的项目是工业规模的，涉及成百上千个程序员，或许并不想让事情过于动态。既要兼顾通用性和威力，又要满足商业产品的需要。让代码能在运行时更换新版这一点很美好，但之后如果出了错则会成为调试者的噩梦。如果数十次动态改动代码，而它随后崩溃了，那么找出准确的错误原因可不是一件容易的事。

PlanetLab里的Erlang

几年前，当我还在做研究的时候，曾经与PlanetLab一起共事。我能访问PlanetLab网络（它是一个全球范围的研究网络：http://www.planet-lab.org），所以我在PlanetLab的所有（大约450台）机器上都安装了“空白”的Erlang服务器。当时我并不知道要拿这些机器来做什么，因此只是设立了服务器架构以供将来使用。

让这层架构运行起来之后，我很容易就能向这些空白服务器发送消息来让它们变成真正的服务器。

举个例子，通常的做法是启用一个Web服务器，然后安装Web服务器插件。我的做法是后退一步，先安装一个空白服务器，以后再让插件把它转变成Web服务器。当我们不再需要Web服务器时，就可以把它变成别的东西。

本节里的服务器示例其实并不怎么正确。它们的编写方式是为了强调有关的概念，然而确实存在着一两个极其微小的隐含错误。我不会马上告诉你这些错误是什么，不过在本章的最后我会给出一些提示。

Erlang的`gen_server`模块是不断强化的服务器（就像在本章里编写的那些）逐渐形成的逻辑成果。

它从1998年起就被用于工业产品。一个产品可以包含数百个服务器，这些服务器正是程序员使用普通的顺序代码编写的。所有的错误处理和非函数行为都被排除在服务器的通用部分之外。

现在我们将跨越想象，来看看真正的`gen_server`。

22.2　gen_server 入门

我将直接把你扔进深水区。以下三点是编写`gen_server`回调模块的简要步骤。

(1) 确定回调模块名。

(2) 编写接口函数。

(3) 在回调模块里编写六个必需的回调函数。

这真的很简单。不要多想，只需按步骤行事！

22.2.1　确定回调模块名

我们将制作一个简单的支付系统。把这个模块称为`my_bank`（我的银行）。

22.2.2 编写接口方法

我们将定义五个接口方法，它们都在my_bank模块里。

- start()
  打开银行。
- stop()
  关闭银行。
- new_account(Who)
  创建一个新账户。
- deposit(Who, Amount)
  把钱存入银行。
- withdraw(Who, Amount)
  把钱取出来（如果有结余的话）。

每个函数都正好对应一个gen_server方法调用，代码如下：

my_bank.erl

```
start() -> gen_server:start_link({local, ?SERVER}, ?MODULE, [], []).
stop()  -> gen_server:call(?MODULE, stop).

new_account(Who)      -> gen_server:call(?MODULE, {new, Who}).
deposit(Who, Amount)  -> gen_server:call(?MODULE, {add, Who, Amount}).
withdraw(Who, Amount) -> gen_server:call(?MODULE, {remove, Who, Amount}).
```


gen_server:start_link({local, Name}, Mod, ...)会启动一个本地服务器。如果第一个参数是原子global，它就会启动一个能被Erlang节点集群访问的全局服务器。start_link的第二个参数是Mod，也就是回调模块名。宏?MODULE会展开成模块名my_bank。目前我们将忽略gen_server:start_link的其他参数。

gen_server:call(?MODULE, Term)被用来对服务器进行远程过程调用。

22.2.3 编写回调方法

我们的回调模块必须导出六个回调方法：init/1、handle_call/3、handle_cast/2、handle_info/2、terminate/2和code_change/3。

为简单起见，可以使用一些模板来制作gen_server。下面是最简单的一种：

gen_server_template.mini

```
-module().
%% gen_server迷你模板
-behaviour(gen_server).
-export([start_link/0]).
%% gen_server回调函数
```

```
-export([init/1, handle_call/3, handle_cast/2, handle_info/2,
         terminate/2, code_change/3]).

start_link() -> gen_server:start_link({local, ?SERVER}, ?MODULE, [], []).
init([]) -> {ok, State}.

handle_call(_Request, _From, State) -> {reply, Reply, State}.
handle_cast(_Msg, State) -> {noreply, State}.
handle_info(_Info, State) -> {noreply, State}.
terminate(_Reason, _State) -> ok.
code_change(_OldVsn, State, Extra) -> {ok, State}.
```

这个模板包含了一套简单的框架，可以填充它们来制作服务器。如果忘记定义合适的回调函数，编译器就会根据关键字-behaviour来生成警告或错误消息。start_link()函数里的服务器名（宏?SERVER）需要进行定义，因为它默认是没有定义的。

提示　如果你正在使用Emacs，那么敲几下按键就能调入一个gen_server模板。如果你在erlang-mode（Erlang模式）下编辑，则可以通过Erlang > Skeletons菜单生成的标签页来创建一个gen_server模板。如果没有Emacs，无需惊慌。我会在本章最后把模板放上来。

我们将从模板入手，对它稍作修改。要做的就是让接口方法里的参数与模板里的参数保持一致。

handle_call/3函数最为重要。我们必须编写代码，让它匹配接口方法里定义的三种查询数据类型。也就是说，必须填写以下代码里的这些点：

```
handle_call({new, Who}, From, State} ->
    Reply  = ...
    State1 = ...
    {reply, Reply, State1};
handle_call({add, Who, Amount}, From, State} ->
    Reply  = ...
    State1 = ...
    {reply, Reply, State1};
handle_call({remove, Who, Amount}, From, State} ->
    Reply  = ...
    State1 = ...
    {reply, Reply, State1};
```

这段代码里的Reply值会作为远程过程调用的返回值发回客户端。

State只是一个代表服务器全局状态的变量，它会在服务器里到处传递。在我们的银行模块里，这个状态永远不会发生变化，它只是一个ETS表的索引，属于常量（虽然表的内容会变化）。

填写模板并稍加改动之后，就形成了以下代码：

my_bank.erl

```
init([]) -> {ok, ets:new(?MODULE,[])}.

handle_call({new,Who}, _From, Tab) ->
```

```
    Reply = case ets:lookup(Tab, Who) of
                 []  -> ets:insert(Tab, {Who,0}),
                        {welcome, Who};
                 [_] -> {Who, you_already_are_a_customer}
            end,
    {reply, Reply, Tab};

handle_call({add,Who,X}, _From, Tab) ->
    Reply = case ets:lookup(Tab, Who) of
                 []  -> not_a_customer;
                 [{Who,Balance}] ->
                     NewBalance = Balance + X,
                     ets:insert(Tab, {Who, NewBalance}),
                     {thanks, Who, your_balance_is,  NewBalance}
            end,
    {reply, Reply, Tab};

handle_call({remove,Who, X}, _From, Tab) ->
    Reply = case ets:lookup(Tab, Who) of
                 []  -> not_a_customer;
                 [{Who,Balance}] when X =< Balance ->
                     NewBalance = Balance - X,
                     ets:insert(Tab, {Who, NewBalance}),
                     {thanks, Who, your_balance_is,  NewBalance};
                 [{Who,Balance}] ->
                     {sorry,Who,you_only_have,Balance,in_the_bank}
            end,
    {reply, Reply, Tab};

handle_call(stop, _From, Tab) ->
    {stop, normal, stopped, Tab}.
handle_cast(_Msg, State) -> {noreply, State}.
handle_info(_Info, State) -> {noreply, State}.
terminate(_Reason, _State) -> ok.
code_change(_OldVsn, State, _Extra) -> {ok, State}.
```

22

调用gen_server:start_link(Name, CallBackMod, StartArgs, Opts)来启动服务器，之后第一个被调用的回调模块方法是Mod:init(StartArgs)，它必须返回{ok, State}。State的值作为handle_call的第三个参数重新出现。

请注意我们是如何停止服务器的。handle_call(stop, From, Tab)返回{stop, normal, stopped, Tab}，它会停止服务器。第二个参数（normal）被用作my_bank:terminate/2的首个参数。第三个参数（stopped）会成为my_bank:stop()的返回值。

就是这样，我们的任务完成了。下面来访问一下银行。

```
1> my_bank:start().
{ok,<0.33.0>}
2> my_bank:deposit("joe", 10).
not_a_customer
```

```
3> my_bank:new_account("joe").
{welcome,"joe"}
4> my_bank:deposit("joe", 10).
{thanks,"joe",your_balance_is,10}
5> my_bank:deposit("joe", 30).
{thanks,"joe",your_balance_is,40}
6> my_bank:withdraw("joe", 15).
{thanks,"joe",your_balance_is,25}
7> my_bank:withdraw("joe", 45).
{sorry,"joe",you_only_have,25,in_the_bank}
```

22.3 gen_server 的回调结构

理解了相关概念之后，我们来详细了解一下gen_server的回调结构。

22.3.1 启动服务器

gen_server:start_link(Name, Mod, InitArgs, Opts)这个调用是所有事物的起点。它会创建一个名为Name的通用服务器，回调模块是Mod，Opts则控制通用服务器的行为。在这里可以指定消息记录、函数调试和其他行为。通用服务器通过调用Mod:init(InitArgs)启动。

图22-1展示了init的模板项（完整的模板可以在A.1节找到）。

```
%%--------------------------------------------------------------------
%% @private
%% @doc
%% 初始化服务器
%%
%% @spec init(Args) -> {ok, State} |
%%                     {ok, State, Timeout} |
%%                     ignore |
%%                     {stop, Reason}
%% @end
%%--------------------------------------------------------------------
init([]) ->
    {ok, #state{}}.
```

图22-1　init的模板项

在通常的操作里，只会返回{ok, State}。要了解其他参数的含义，请参考gen_server的手册页。

如果返回{ok, State}，就说明我们成功启动了服务器，它的初始状态是State。

22.3.2 调用服务器

要调用服务器，客户端程序需要执行gen_server:call(Name, Request)。它最终调用的

是回调模块里的handle_call/3。

handle_call/3的模板项如下：

```
%%--------------------------------------------------------------------
%% @private
%% @doc
%% 处理调用消息
%%
%% @spec handle_call(Request, From, State) ->
%%                                   {reply, Reply, State} |
%%                                   {reply, Reply, State, Timeout} |
%%                                   {noreply, State} |
%%                                   {noreply, State, Timeout} |
%%                                   {stop, Reason, Reply, State} |
%%                                   {stop, Reason, State}
%% @end
%%--------------------------------------------------------------------
handle_call(_Request, _From, State) ->
    Reply = ok,
    {reply, Reply, State}.
```

Request（gen_server:call/2的第二个参数）作为handle_call/3的第一个参数重新出现。From是发送请求的客户端进程的PID，State则是客户端的当前状态。

我们通常会返回{reply, Reply, NewState}。在这种情况下，Reply会返回客户端，成为gen_server:call的返回值。NewState则是服务器接下来的状态。

其他的返回值（{noreply, ..}和{stop, ..}）相对不太常用。no reply会让服务器继续工作，但客户端会等待一个回复，所以服务器必须把回复的任务委派给其他进程。用适当的参数调用stop会停止服务器。

22

22.3.3 调用和播发

我们已经见过了gen_server:call和handle_call之间的交互，它的作用是实现远程过程调用。gen_server:cast(Name, Msg)则实现了一个播发（cast），也就是没有返回值的调用（实际上就是一个消息，但习惯上称它为播发来与远程过程调用相区分）。

对应的回调方法是handle_cast，它的模板项如下：

```
%%--------------------------------------------------------------------
%% @private
%% @doc
%% 处理播发消息
%%
%% @spec handle_cast(Msg, State) -> {noreply, State} |
%%                                  {noreply, State, Timeout} |
%%                                  {stop, Reason, State}
%% @end
%%--------------------------------------------------------------------
```

```
handle_cast(_Msg, State) ->
    {noreply, State}.
```

这个处理函数通常只返回{noreply, NewState}或{stop, ...}。前者改变服务器的状态，后者停止服务器。

22.3.4　发给服务器的自发性消息

回调函数handle_info(Info, State)被用来处理发给服务器的自发性消息。自发性消息是一切未经显式调用gen_server:call或gen_server:cast而到达服务器的消息。举个例子，如果服务器连接到另一个进程并捕捉退出信号，就可能会突然收到一个预料之外的{'EXIT', Pid, What}消息。除此之外，系统里任何知道通用服务器PID的进程都可以向它发送消息。这样的消息在服务器里表现为info值。

handle_info的模板项如下：

```
%%--------------------------------------------------------------------
%% @private
%% @doc
%% 处理所有非调用/播发的消息
%%
%% @spec handle_info(Info, State) -> {noreply, State} |
%%                                   {noreply, State, Timeout} |
%%                                   {stop, Reason, State}
%% @end
%%--------------------------------------------------------------------
handle_info(_Info, State) ->
    {noreply, State}.
```

它的返回值和handle_cast相同。

22.3.5　后会有期，宝贝

服务器会因为许多原因而终止。某个以handle_开头的函数也许会返回一个{stop, Reason, NewState}，服务器也可能崩溃并生成{'EXIT', reason}。在所有这些情况下，无论它们是怎样发生的，都会调用terminate(Reason, NewState)。它的模板项如下。

```
%%--------------------------------------------------------------------
%% @private
%% @doc
%% 这个函数是在某个gen_server即将终止时调用的。
%% 它应当是Module:init/1的逆操作，并进行必要的清理。
%% 当它返回时，<mod>gen_server</mod>终止并生成原因Reason。
%% 它的返回值会被忽略。
%%
%% @spec terminate(Reason, State) -> void()
%% @end
```

```
%%--------------------------------------------------------------------
terminate(_Reason, _State) ->
    ok.
```

这段代码不能返回一个新状态，因为我们已经终止了。但是了解服务器在终止时的状态非常有用。可以把状态保存到磁盘，把它放入消息发送给别的进程，或者根据应用程序的意愿丢弃它。如果想让服务器过后重启，就必须编写一个“我胡汉三又回来了”的函数，由terminate/2触发。

22.3.6　代码更改

你可以在服务器运行时动态更改它的状态。这个回调函数会在系统执行软件升级时由版本处理子系统调用。

OTP系统文档[①]里的版本处理一节详细介绍了这个主题。

```
%%--------------------------------------------------------------------
%% @private
%% @doc
%% 在代码更改时转换进程状态
%%
%% @spec code_change(OldVsn, State, Extra) -> {ok, NewState}
%% @end
%%--------------------------------------------------------------------
code_change(_OldVsn, State, _Extra) ->
    {ok, State}.
```

22

22.4　填写 gen_server 模板

编写OTPgen_server大致上就是用你的代码填充一个预制模板，下面是一个例子。前一节分别列出了gen_server的各个区块。Emacs内建了gen_server模板，但如果你不使用Emacs，也可以在A.1节找到完整的模板。

我通过填充模板生成了一个名为my_bank的银行模块。这段代码取自模板。我移除了模板里的所有注释，这样就能清楚地看到代码的结构。

my_bank.erl

```
-module(my_bank).

-behaviour(gen_server).
-export([start/0]).
%% gen_server回调函数
-export([init/1, handle_call/3, handle_cast/2, handle_info/2,
         terminate/2, code_change/3]).
```

① http://www.erlang.org/doc/pdf/otp-system-documentation.pdf

```
-compile(export_all).
-define(SERVER, ?MODULE).

start() -> gen_server:start_link({local, ?SERVER}, ?MODULE, [], []).
stop()  -> gen_server:call(?MODULE, stop).

new_account(Who)      -> gen_server:call(?MODULE, {new, Who}).
deposit(Who, Amount)  -> gen_server:call(?MODULE, {add, Who, Amount}).
withdraw(Who, Amount) -> gen_server:call(?MODULE, {remove, Who, Amount}).

init([]) -> {ok, ets:new(?MODULE,[])}.

handle_call({new,Who}, _From, Tab) ->
    Reply = case ets:lookup(Tab, Who) of
                []  -> ets:insert(Tab, {Who,0}),
                       {welcome, Who};
                [_] -> {Who, you_already_are_a_customer}
            end,
    {reply, Reply, Tab};

handle_call({add,Who,X}, _From, Tab) ->
    Reply = case ets:lookup(Tab, Who) of
                []  -> not_a_customer;
                [{Who,Balance}] ->
                    NewBalance = Balance + X,
                    ets:insert(Tab, {Who, NewBalance}),
                    {thanks, Who, your_balance_is,  NewBalance}
            end,
    {reply, Reply, Tab};

handle_call({remove,Who, X}, _From, Tab) ->
    Reply = case ets:lookup(Tab, Who) of
                []  -> not_a_customer;
                [{Who,Balance}] when X =< Balance ->
                    NewBalance = Balance - X,
                    ets:insert(Tab, {Who, NewBalance}),
                    {thanks, Who, your_balance_is,  NewBalance};
                [{Who,Balance}] ->
                    {sorry,Who,you_only_have,Balance,in_the_bank}
            end,
    {reply, Reply, Tab};

handle_call(stop, _From, Tab) ->
    {stop, normal, stopped, Tab}.
handle_cast(_Msg, State) -> {noreply, State}.
handle_info(_Info, State) -> {noreply, State}.
terminate(_Reason, _State) -> ok.
code_change(_OldVsn, State, _Extra) -> {ok, State}.
```

22.5　深入探索

gen_server其实相当简单。我们并未全面介绍gen_server里的接口函数，也没有解释这些函数的全部变量。一旦理解了基本的概念，就可以去gen_server的手册页里查找更多细节。

这一章只介绍了最简单的gen_server使用方式，但它应该足以应付大多数需求了。复杂程度更高的应用程序经常会让gen_server回复一个noreply返回值，并把真正的回复任务委派给另一个进程。要了解更多这方面的信息，请阅读"Design Principles"①（设计原则）文档，以及sys和proc_lib模块的手册页。

本章介绍了把服务器行为抽象成两个部分这一概念：通用部分可以用于所有服务器，特有部分（处理模块）可以用来对通用部分进行定制。这么做的主要优点是代码能整齐地一分为二。通用部分解决众多并发和错误处理问题，而处理模块只包含顺序代码。

在此之后，我们介绍了OTP系统里的第一种主要行为：gen_server，并展示了如何从一个相当简单且容易理解的服务器入手，通过逐步的转变来实现它。

gen_server的用途很广，但它并不能包治百病。gen_server的客户端-服务器交互模式有时候会让人感觉别扭，与你的问题不能良好兼容。如果是这样，就需要重新思考制作gen_server所需要的转变步骤，根据问题的特殊需要来修改它们。

当我们从单个服务器转向系统时，就会用到很多服务器。我们希望能以一致的方式监视它们、重启退出的服务器以及记录错误。这就是下一章的主题。

22.6　练习

在下面这些练习里，我们将用job_centre模块制作一个服务器，它用gen_server实现一种任务管理服务。任务中心（job center）持有一个必须完成的任务队列，这些任务会被编号，任何人都能向队列添加任务。工人可以从队列请求任务，并告诉任务中心已经执行了某项任务。任务是由fun表示的，要执行任务F，工人必须执行F()函数。

(1) 实现任务中心的基本功能，它的接口如下。

- `job_centre:start_link() -> true`
  启动任务中心服务器。
- `job_centre:add_job(F) -> JobNumber`
  添加任务F到任务队列，然后返回一个整数任务编号。
- `job_centre:work_wanted() -> {JobNumber,F} | no`
  请求任务。如果工人想要一个任务，就调用job_centre:work_wanted()。如果队列里有任务，就会返回一个{JobNumber, F}元组。工人执行F()来完成任务。如果队列里没

① http://www.erlang.org/doc/pdf/design_principles.pdf

有任务，则会返回no。请确保同一项任务每次只分配给一个工人，并确保系统是公平的，意思是任务按照请求的顺序进行分配。

- job_centre:job_done(JobNumber)

  发出任务完成的信号。如果工人完成了某一项任务，就必须调用job_centre:job_done(JobNumber)。

(2) 添加一个名为job_centre:statistics()的统计函数，让它报告队列内、进行中和已完成任务的状态。

(3) 添加监视工人进程的代码。如果某个工人进程挂了，请确保它所执行的任务被返回到等待完成的任务池里。

(4) 检查是否有懒惰的工人，也就是接受工作但不按时完成的进程。把任务请求函数修改为返回{JobNumber, JobTime, F}，其中JobTime是工人必须完成任务的秒数。如果工人在JobTime - 1时还未完成任务，服务器就应当向其发送一个hurry_up（快点儿）消息，而在JobTime + 1时应该用调用exit(Pid, youre_fired)（你被解雇了）来杀掉这个工人进程。

(5) 可选练习：实现一个工会服务器来监督工人的权利，防止他们没收到警告就被解雇。提示：使用进程跟踪基本函数来实现它。

第 23 章

用OTP构建系统

这一章将构建一个系统，把它作为一家网络公司的后端。我们这家公司销售两种商品：质数和面积。顾客可以从这里购买质数，也可以请我们计算某个几何对象的面积。我认为公司发展潜力巨大。

将制作两个服务器：一个生成质数，另一个计算面积。并使用22.2节里讨论的`gen_server`框架来实现它们。

在构建系统时必须考虑到错误。即使彻底测试了软件，也不一定能捕获所有的bug。假设其中一个服务器带有会导致服务器崩溃的致命错误。确切地说，故意引入一个错误来使其中一个服务器崩溃。

当服务器崩溃时，需要一种机制来检测这种情况并重启它，为此将用到监控树（supervision tree）这个概念。创建一个监控器来管理服务器，如果服务器崩溃就重启它们。

当然，如果服务器确实崩溃了，我们希望知道它崩溃的原因，这样就能在未来修复这个问题。为了记录所有错误，可以使用OTP的错误记录器。我们会展示如何配置错误记录器，以及如何根据错误日志生成错误报告。

计算质数（特别是大质数）时，CPU可能会过热，这就需要开启一个强力风扇来避免这种情况。要做到这一点，需要考虑警报。我们会用OTP事件处理框架来生成和处理警报。

所有这些主题（创建服务器、监控服务器、记录错误和检测警报）是一切生产系统都必须解决的典型问题。因此，虽然公司前途未卜，却可以在许多系统里重复使用这种架构。事实上，许多在商业上获得成功的公司都在使用这种架构。

最后，当一切都能正常工作时，所有代码都会被打包到一个OTP应用程序里。这是一种把围绕某个问题的事物组合到一起的专用方法，让OTP系统自身来启动、停止和管理它。

要决定按什么顺序呈现这些材料并不容易，因为许多领域之间存在循环依赖关系。错误记录只是事件管理的一个特例。警报就是一种消息，错误记录器是一个被监控进程，但是进程监控器可以调用错误记录器。

我将尝试理出某种顺序，把这些主题相对有条理地呈现出来。我们将做下面这些事。

(1) 熟悉通用事件处理器里用到的概念。

(2) 了解错误记录器的工作方式。

(3) 添加警报管理功能。

(4) 编写两个应用程序服务器。

(5) 制作一个监控树，并给它添加服务器。

(6) 把这一切打包成一个应用程序。

23.1　通用事件处理

事件就是已发生的事情：它是值得注意的，程序员认为有人应该对它做些什么。

如果在编程的时候发生了一件值得注意的事，就会发送一个event消息给某个注册进程，就像这样：

```
RegProcName ! {event, E}
```

E是事件（可以是任意Erlang数据类型），RegProcName是注册进程名。

发送消息后我们不知道（也不关心）它的命运。只是完成自己的任务，告诉其他人有什么事发生。

现在把注意力转向接收事件消息的进程，它被称为事件处理器。最简单的事件处理器就是一个“什么都不做”的处理器。当它收到一个{event, X}消息时不会对它做任何处理，只会把它丢弃。

下面是我们对通用事件处理程序的首次尝试。

event_handler.erl

```
-module(event_handler).
-export([make/1, add_handler/2, event/2]).
%% 制作一个名为Name的新事件处理器
%% 处理函数是no_op，代表不对事件做任何处理
make(Name) ->
    register(Name, spawn(fun() -> my_handler(fun no_op/1) end)).
add_handler(Name, Fun) -> Name ! {add, Fun}.

%% 生成一个事件
event(Name, X) -> Name ! {event, X}.

my_handler(Fun) ->
    receive
        {add, Fun1} ->
            my_handler(Fun1);
        {event, Any} ->
            (catch Fun(Any)),
            my_handler(Fun)
    end.
no_op(_) -> void.
```

这个事件处理器的API如下。

- ❑ `event_handler:make(Name)`
  制作一个“什么都不干”的事件处理器Name（一个原子）。这样消息就有地方发送了。
- ❑ `event_handler:event(Name, X)`
  发送消息X到名为Name的事件处理器。
- ❑ `event_handler:add_handler(Name, Fun)`
  给名为Name的事件处理器添加一个处理函数Fun。这样当事件X发生时，事件处理器就会执行Fun(X)。

现在创建一个事件处理器并生成一个错误。

```
1> event_handler:make(errors).
true
2> event_handler:event(errors, hi).
{event,hi}
```

没什么特别的事发生，因为我们还没有给事件处理器安装回调模块。

要让事件处理器能做点什么，必须编写一个回调模块并把它安装到事件处理器里。这是一个事件处理器回调模块的代码：

motor_controller.erl

```
-module(motor_controller).
-export([add_event_handler/0]).

add_event_handler() ->
    event_handler:add_handler(errors, fun controller/1).
controller(too_hot) ->
    io:format("Turn off the motor~n");
controller(X) ->
    io:format("~w ignored event: ~p~n",[?MODULE, X]).
```

编译之后就可以安装它了。

```
3> c(motor_controller).
{ok,motor_controller}
4> motor_controller:add_event_handler().
{add,#Fun<motor_controller.0.99476749>}
```

现在当我们发送消息给处理器时，函数`motor_controller:controller/1`会处理这些消息。

```
5> event_handler:event(errors, cool).
motor_controller ignored event: cool
{event,cool}
6> event_handler:event(errors, too_hot).
Turn off the motor
{event,too_hot}
```

这个练习有两个目的。首先，提供一个名称来作为消息发送的目的地，也就是名为errors

的注册进程。然后定义一个协议来发送事件给这个注册进程，但并没有说明消息到达后会发生什么。事实上，唯一发生的事就是执行了`no_Op(X)`。随后安装一个自定义的事件处理器，这么做本质上是把事件生成和事件处理分开进行，这样我们就能暂不决定如何处理事件，同时又不影响事件生成。

这里的要点在于事件处理器提供了一种架构，让我们可以安装自定义的处理器。

错误记录器的架构遵循事件处理器的模式。可以在错误记录器里安装不同的处理器来让它做不同的事情。警报处理架构同样遵循这一模式。

23.2　错误记录器

OTP系统自带一个可定制的错误记录器。可以从三个角度看待错误记录器：程序员视角关心程序员为了记录错误而在代码里中做的函数调用；配置视角关心错误记录器在何处以及如何保存数据；报告视角关心对已发生错误的分析。稍后会对这三个角度逐一进行讨论。

“可以改主意”的超后期绑定

假设我们在编写一个对程序员隐藏`event_handler:event`方法的函数，例如下面这个：

lib_misc.erl

```
too_hot() ->
event_handler:event(errors, too_hot).
```

然后告诉程序员在出问题时要调用代码里的`lib_misc:too_hot()`。在大多数编程语言里，对函数`too_hot`的调用会被静态或动态链接到调用该函数的代码上。一旦链接完成，它就会根据代码要求执行固定任务。如果我们后来改变主意要做其他的事，就没有什么简单的办法能改变系统行为了。

Erlang处理事件的方法则完全不一样。它允许我们把事件生成和事件处理分开进行。任何时候都可以修改处理方法，只需向事件处理器发送一个新的处理函数即可。不存在什么静态链接，只要你愿意，事件处理函数可以随时更换。

通过这种机制，可以构建与时俱进的系统，永远不需要停止它们来升级代码。

注意：这不是“后期绑定”，而是“超后期绑定，以后还可以改主意”。

23.2.1　记录错误

对程序员来说，这个错误记录器的API很简单。以下是一个简单的API子集：

- -spec error_logger:error_msg(String) -> ok
  向错误记录器发送一个错误消息。

```
1> error_logger:error_msg("An error has occurred\n").
=ERROR REPORT==== 15-Jul-2013::17:03:14 ===
An error has occurred
ok
```

- -spec error_logger:error_msg(Format, Data) -> ok

  向错误记录器发送一个错误消息。它的参数和`io:format(Format, Data)`相同。

```
2> error_logger:error_msg("~s, an error has occurred\n", ["Joe"]).
=ERROR REPORT==== 15-Jul-2013::17:04:03 ===
Joe, an error has occurred
ok
```

- -spec error_logger:error_report(Report) -> ok

  向错误记录器发送一个标准错误报告。

  - `-type Report = [{Tag, Data} | term() | string() ].`
  - `-type Tag = term().`
  - `-type Data = term().`

```
3> error_logger:error_report([{tag1,data1},a_term,{tag2,data}]).
=ERROR REPORT==== 15-Jul-2013::17:04:41 ===
tag1: data1
a_term
tag2: data
```

这只是全部可用API的一个子集。详细讨论它们意义不大，我们的程序只会用到`error_msg`。完整的细节可以在`error_logger`的手册页里找到。

23.2.2　配置错误记录器

有多种方式可用来配置错误记录器。可以让Erlang shell显示所有错误（如果没有特别设置的话就是默认值），也可以把shell里报告的所有错误写入一个格式化文本文件。最后，还可以创建一个*滚动日志*（rotating log）。可以把滚动日志看作是一个大型循环缓冲区，内含错误记录器生成的消息。新消息进来后会被附加到日志的末尾，如果日志满了，最早的条目就会被删除。

滚动日志极其有用。你决定日志应当占据多少个文件，以及每个日志文件能有多大，然后系统负责在一个大型循环缓冲区里删除旧日志文件和创建新文件。可以调整日志的大小来保存最近几天的操作记录，这通常足以应付大多数用途了。

1. 标准错误记录器

在启动Erlang时可以给系统提供一个*启动参数*。

- `$ erl -boot start_clean`

  它会创建一个适合进行程序开发的环境，只提供一种简单的错误记录形式。（不带启动参数的`erl`命令就等于`erl -boot start_clean`。）

- `$ erl -boot start_sasl`
  它会创建一个适合运行生产系统的环境。系统架构支持库（System Architecture Support Libraries，简称SASL）将负责错误记录和过载保护等工作。

日志文件的配置最好通过配置文件实现，因为没人能记住记录器的全部参数。在接下来的几节里，我们将来看看默认系统的工作方式，以及改变错误记录器工作方式的四种特定配置。

2. 无配置SASL

当我们不带配置文件启动SASL时会发生下面这些事：

```
$ erl -boot start_sasl
Erlang R16B (erts-5.10.1) [source] [smp:2:2] ...

=PROGRESS REPORT==== 26-May-2013::12:48:37 ===
supervisor: {local,sasl_safe_sup}
   started: [{pid,<0.35.0>},
             {name,alarm_handler},
             {mfargs,{alarm_handler,start_link,[]}},
             {restart_type,permanent},
             {shutdown,2000},
             {child_type,worker}]
... many lines removed ...
Eshell V5.10.1  (abort with ^G)
```

现在调用`error_logger`里的某个方法来报告一个错误。

```
1> error_logger:error_msg("This is an error\n").
=ERROR REPORT==== 26-May-2013::12:54:03 ===
This is an error
ok
```

请注意，错误是在Erlang shell里报告的。错误的报告位置由错误记录器配置决定。

3. 控制记录内容

错误记录器会生成多种报告类型。

- 监控器报告
  这些报告会在OTP监控器启动或停止被监控进程时生成（参见23.5节）。
- 进度报告
  这些报告会在OTP监控器启动或停止时生成。
- 崩溃报告
  如果某个被OTP行为启动的进程因为`normal`或`shutdown`以外的原因终止，这些报告就会生成。

这三种报告会自动生成，程序员无须做任何事。

另外，还可以显式调用`error_logger`模块里的方法来生成三种类型的日志报告。这让我们能够记录错误、警告和信息消息。这三个名词没什么语义含义，只是一些标签，程序员用它们来提示错误日志条目的性质。

后面分析错误日志时，可以用这些标签来帮助我们决定该调查哪些日志条目。在配置错误记录器时可以选择只保存错误，丢弃其他所有类型的条目。现在，编写配置文件elog1.config来配置错误记录器。

elog1.config

```
%% 无tty
[{sasl, [
         {sasl_error_logger, false}
        ]}].
```

如果用这个配置文件启动系统，就只会得到错误报告，不会有进度和其他报告。所有这些错误报告只会出现在shell里。

```
$ erl -boot start_sasl -config elog1
1> error_logger:error_msg("This is an error\n").
=ERROR REPORT==== 15-Jul-2013::11:53:08 ===
This is an error
ok
```

4. 文本文件和shell

接下来的配置文件会在shell里列出错误报告，所有的进度报告则会保存在一个文件里。

elog2.config

```
%% 单文本文件，最小化tty

[{sasl, [
         %% 所有的报告都写入这个文件
         {sasl_error_logger, {file, "/Users/joe/error_logs/THELOG"}}
        ]}].
```

要测试它，我们可以启动Erlang，生成一个错误消息，然后查看日志文件。

```
$ erl -boot start_sasl -config elog2
Erlang R16B (erts-5.10.1) [source] [smp:2:2] ...
Eshell V5.10.1  (abort with ^G)
1> error_logger:error_msg("This is an error\n").
=ERROR REPORT==== 26-May-2013::13:07:46 ===
This is an error
ok
```

如果现在查看/Users/joe/error_logs/THELOG，就会发现它的开头如下：

```
=PROGRESS REPORT==== 15-Jul-2013::11:30:55 ===
supervisor: {local,sasl_safe_sup}
   started: [{pid,<0.34.0>},
             {name,alarm_handler},
             {mfa,{alarm_handler,start_link,[]}},
             {restart_type,permanent},
             {shutdown,2000},
```

```
              {child_type,worker}]
...
```

这里只列出了进度报告，而它们原本应该出现在shell里。进度报告是关于大事件的，比如启动和停止应用程序。但是error_logger:error_msg/1报告的错误没有保存在日志里，为此我们必须配置一个滚动日志。

5. 滚动日志和shell

下面的配置既能提供shell输出，又能把写入shell的所有信息复制到一个滚动日志文件里。

elog3.config

```
%% 滚动日志和最小化tty
[{sasl, [
         {sasl_error_logger, false},
         %% 定义滚动日志的参数
         %% 日志文件目录
         {error_logger_mf_dir,"/Users/joe/error_logs"},
         %% # 每个日志文件的字节数
         {error_logger_mf_maxbytes,10485760}, % 10 MB
         %% 日志文件的最大数量
         {error_logger_mf_maxfiles, 10}
        ]}].
```

```
erl -boot start_sasl -config elog3
Erlang R16B (erts-5.10.1) [source] [smp:2:2] ...
Eshell V5.10.1  (abort with ^G)
1> error_logger:error_msg("This is an error\n").
2>
=ERROR REPORT==== 26-May-2013::13:14:31 ===
This is an error
```

这个日志的最大文件大小是10MB，达到10MB时会回绕或者“滚动”。可想而知，这是一个非常有用的配置。 运行系统时，所有的错误都会写入一个滚动错误日志。我们会在本章后面看到如何从日志里提取错误。

6. 生产环境

在生产环境里，我们真正感兴趣的只有错误，而非进度或信息报告，所以只让错误记录器报告错误。如果没有这个设置，系统也许就会被信息和进度报告所淹没。

elog4.config

```
%% 滚动日志和错误
[{sasl, [
         %% 最小化shell错误记录
         {sasl_error_logger, false},
         %% 只报告错误
         {errlog_type, error},
         %% 定义滚动日志的参数
         %% 日志文件目录
```

```
 {error_logger_mf_dir,"/Users/joe/error_logs"},
 %% 每个日志文件的字节数
 {error_logger_mf_maxbytes,10485760}, % 10 MB
 %% 日志文件的最大数量
 {error_logger_mf_maxfiles, 10}
]}].
```

运行它会产生与之前例子类似的输出，区别在于错误日志里只会有错误报告。

23.2.3 分析错误

阅读错误日志是rb模块的责任，它的接口极其简单。

```
$ erl -boot start_sasl -config elog3
...
1> rb:help().
Report Browser Tool - usage
===========================
2> rb:start()         - start the rb_server with default options
rb:start(Options)  - where Options is a list of:
                     {start_log, FileName}
                       - default: standard_io
                     {max, MaxNoOfReports}
                       - MaxNoOfReports should be an integer or 'all'
                       - default: all
    ...
    ... many lines omitted ...
    ...
3> rb:start([{max,20}]).
rb: reading report...done.
```


首先必须用正确的配置文件启动Erlang，这样才能定位错误日志；然后启动报告浏览器，告诉它要读取多少日志条目（在这个案例里是最后20条）。现在列出日志里的条目。

```
4> rb:list().
No                Type   Process       Date     Time
==                ====   =======       ====     ====
12            progress  <0.31.0> 2013-05-26 13:21:53
11            progress  <0.31.0> 2013-05-26 13:21:53
10            progress  <0.31.0> 2013-05-26 13:21:53
9             progress  <0.24.0> 2013-05-26 13:21:53
8                error  <0.25.0> 2013-05-26 13:23:04
7             progress  <0.31.0> 2013-05-26 13:23:58
6             progress  <0.31.0> 2013-05-26 13:24:13
5             progress  <0.31.0> 2013-05-26 13:24:13
4             progress  <0.31.0> 2013-05-26 13:24:13
3             progress  <0.31.0> 2013-05-26 13:24:13
2             progress  <0.24.0> 2013-05-26 13:24:13
1             progress  <0.31.0> 2013-05-26 13:24:17
ok
```

调用error_logger:error_msg/1产生的错误日志条目消息最终成为日志里的第8条。可以像这样检查它：

```
> rb:show(8).
ERROR REPORT  <0.44.0>                   2013-05-26 13:23:04
===========================================================
This is an error
ok
```

要分离出某个错误，可以使用rb:grep(RegExp)这样的命令，它会找出所有匹配正则表达式RegExp的报告。我不想详尽介绍如何分析错误日志，最好的做法是花一点时间与rb交互来看看它能做什么。请注意，实际上永远不需要删除任何一个错误报告，因为滚动机制最终会删除那些老的错误报告。

如果想保留所有的错误日志，就必须定期轮询错误报告并移除你感兴趣的信息。

23.3　警报管理

我们编写的应用程序只需要一个警报，这个警报会在CPU因为计算超大质数而开始熔化时抛出（别忘了我们正在建设一家销售质数的公司）。这次将使用真正的OTP警报处理器（而不是在本章开头看到的简单版）。

这个警报处理器是OTPgen_event行为的回调模块，它的代码如下。

my_alarm_handler.erl

```
-module(my_alarm_handler).
-behaviour(gen_event).

%% gen_event回调函数
-export([init/1, code_change/3, handle_event/2, handle_call/2,
         handle_info/2, terminate/2]).

%% init(Args)必须返回{ok, State}
init(Args) ->
    io:format("*** my_alarm_handler init:~p~n",[Args]),
    {ok, 0}.

handle_event({set_alarm, tooHot}, N) ->
    error_logger:error_msg("*** Tell the Engineer to turn on the fan~n"),
    {ok, N+1};
handle_event({clear_alarm, tooHot}, N) ->
    error_logger:error_msg("*** Danger over. Turn off the fan~n"),
    {ok, N};
handle_event(Event, N) ->
    io:format("*** unmatched event:~p~n",[Event]),
    {ok, N}.
```

```
handle_call(_Request, N) -> Reply = N, {ok, Reply,  N}.
handle_info(_Info, N)    -> {ok, N}.

terminate(_Reason, _N)   -> ok.
code_change(_OldVsn, State, _Extra) -> {ok, State}.
```

这段代码非常像之前在22.3节里看到的`gen_server`回调代码。其中值得注意的方法是`handle_event(Event, State)`，它应当返回`{ok, NewState}`。`Event`是一个`{EventType, Event-Arg}`形式的元组，其中`EventType`是`set_event`或`clear_event`，而`EventArg`是一个用户提供的参数。稍后会看到这些事件是如何生成的。

现在来找点乐子：启动系统，生成一个警报，安装警报处理器，再生成一个警报……

```
$ erl -boot start_sasl -config elog3
1> alarm_handler:set_alarm(tooHot).
ok
=INFO REPORT==== 15-Jul-2013::14:20:06 ===
alarm_handler: {set,tooHot}
2> gen_event:swap_handler(alarm_handler,
                          {alarm_handler, swap},
                          {my_alarm_handler, xyz}).
*** my_alarm_handler init:{xyz,{alarm_handler,[tooHot]}}
3> alarm_handler:set_alarm(tooHot).
ok
=ERROR REPORT==== 15-Jul-2013::14:22:19 ===
*** Tell the Engineer to turn on the fan
4> alarm_handler:clear_alarm(tooHot).
ok
=ERROR REPORT==== 15-Jul-2013::14:22:39 ===
*** Danger over. Turn off the fan
```

23

刚才发生了以下这些事情。

(1) 用`-boot start_sasl`启动了Erlang。这么做就得到了一个标准警报处理器。当我们设置或清除警报时，什么事都不会发生。这就类似于之前讨论过的“什么都不干”的事件处理器。

(2) 设置警报（第1行）后只得到了一个信息报告。这个警报没有得到特别处理。

(3) 安装了一个自定义警报处理器（第2行）。`my_alarm_handler`的参数（`xyz`）没什么特殊含义，只不过语法要求这里有一个值。但因为我们没有用值而是用了原子`xyz`，所以能在参数打印出来时识别它。

`** my_alarm_handler_init: ...`这段打印输出来自我们的回调模块。

(4) 设置并清除了一个`tooHot`警报（第3和第4行）。自定义警报处理器对其进行了处理，shell打印输出能证明这一点。

读取日志

让我们回到错误记录器里去看看发生了什么。

```
1> rb:start([{max,20}]).
rb: reading report...done.
2> rb:list().
No                Type   Process  Date       Time
==                ====   =======  ====       ====
...
3         info_report    <0.29.0> 2013-07-30 14:20:06
2               error    <0.29.0> 2013-07-30 14:22:19
1               error    <0.29.0> 2013-07-30 14:22:39
3> rb:show(1).

ERROR REPORT  <0.33.0>           2013-07-30 14:22:39
=====================================================
*** Danger over. Turn off the fan
ok
4> rb:show(2).
ERROR REPORT  <0.33.0>           2013-07-30 14:22:19
=====================================================
*** Tell the Engineer to turn on the fan
```

可以看到错误记录机制运行正常。

在实践中，我们会确保错误日志大到足够支持几天或几周的运作。每隔几天（或几周）就会检查错误日志并调查所有错误。

> **注意**　rb模块里有一些函数能选择特定类型的错误或把它们提取到文件里。因此，分析错误日志的过程可以实现完全自动化。

23.4　应用程序服务器

我们的应用程序有两个服务器：一个质数服务器和一个面积服务器。

23.4.1　质数服务器

这里是质数服务器的代码，它是用gen_server行为编写的（参见22.2节）。请注意它是如何纳入我们在前一节中开发的警报处理函数的。

prime_server.erl

```
-module(prime_server).

-behaviour(gen_server).

-export([new_prime/1, start_link/0]).

%% gen_server回调函数
-export([init/1, handle_call/3, handle_cast/2, handle_info/2,
         terminate/2, code_change/3]).
```

```
start_link() ->
    gen_server:start_link({local, ?MODULE}, ?MODULE, [], []).

new_prime(N) ->
    %% 20000（ms）是一个超时设置
    gen_server:call(?MODULE, {prime, N}, 20000).
init([]) ->
    %% 请注意，如果想让terminate/2
    %% 在应用程序停止时被调用，就必须
    %% 设置trap_exit = true

    process_flag(trap_exit, true),
    io:format("~p starting~n",[?MODULE]),
    {ok, 0}.

handle_call({prime, K}, _From, N) ->
    {reply, make_new_prime(K), N+1}.

handle_cast(_Msg, N)  -> {noreply, N}.

handle_info(_Info, N)  -> {noreply, N}.

terminate(_Reason, _N) ->
    io:format("~p stopping~n",[?MODULE]),
    ok.

code_change(_OldVsn, N, _Extra) -> {ok, N}.

make_new_prime(K) ->
    if
        K > 100 ->
            alarm_handler:set_alarm(tooHot),
            N = lib_primes:make_prime(K),
            alarm_handler:clear_alarm(tooHot),
            N;
        true ->
            lib_primes:make_prime(K)
    end.
```

23.4.2 面积服务器

现在轮到面积服务器了，它也是用gen_server行为编写的。请注意，用这种方式编写服务器速度极快。我在编写这个示例时剪切粘贴了质数服务器里的代码，然后把它转变成面积服务器。这只用了几分钟时间。

这个面积服务器不是全世界最有才的程序，而且它还包含一个故意设置的错误（你能找到吗？）。我的计划不算巧妙，就是为了让服务器崩溃然后被监控器重启。此外，我们还会在错误日志里获得关于这一切的报告。

23

area_server.erl

```
-module(area_server).
-behaviour(gen_server).

-export([area/1, start_link/0]).

%% gen_server回调函数
-export([init/1, handle_call/3, handle_cast/2, handle_info/2,
         terminate/2, code_change/3]).

start_link() ->
    gen_server:start_link({local, ?MODULE}, ?MODULE, [], []).

area(Thing) ->
    gen_server:call(?MODULE, {area, Thing}).

init([]) ->
    %% 请注意，如果想让terminate/2
    %% 在应用程序停止时被调用，就必须
    %% 设置trap_exit = true
    process_flag(trap_exit, true),
    io:format("~p starting~n",[?MODULE]),
    {ok, 0}.

handle_call({area, Thing}, _From, N) -> {reply, compute_area(Thing), N+1}.

handle_cast(_Msg, N)  -> {noreply, N}.

handle_info(_Info, N)  -> {noreply, N}.

terminate(_Reason, _N) ->
    io:format("~p stopping~n",[?MODULE]),
    ok.

code_change(_OldVsn, N, _Extra) -> {ok, N}.

compute_area({square, X})       -> X*X;
compute_area({rectongle, X, Y}) -> X*Y.
```

我们已经编写完了应用程序代码，还加了一个小错误。现在必须设立一个监控结构来检测和纠正所有可能在运行时发生的错误。

23.5　监控树

监控树是一种由进程组成的树形结构。树的上级进程（监控器）监视着下级进程（工作器），如果下级进程挂了就会重启它们。监控树有两种，如图23-1所示。

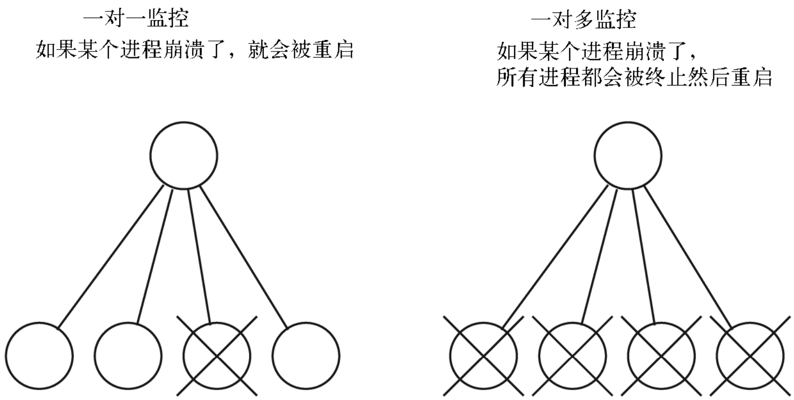


图　23-1

- 一对一监控树

  在一对一监控里，如果某个工作器崩溃了，就会被监控器重启。

- 一对多监控树

  在一对多监控里，如果任何一个工作器崩溃了，所有工作进程都会被终止（通过调用相应回调模块里的`terminate/2`函数）然后重启。

监控器是用OTP supervisor行为创建的。这个行为用一个回调模块作为参数，里面指定了监控策略以及如何启动监控树里的各个工作进程。监控树通过以下形式的函数指定：

```
init(...) ->
    {ok, {
          {RestartStrategy, MaxRestarts, Time},
          [Worker1, Worker2, ...]
         }}.
```

这里的`RestartStrategy`是原子`one_for_one`或`one_for_all`，`MaxRestarts`和`Time`则指定“重启频率”。如果一个监控器在`Time`秒内执行了超过`MaxRestarts`次重启，那么这个监控器就会终止所有工作进程然后退出。这是为了防止出现一种情形，即某个进程崩溃、被重启，然后又因为相同原因崩溃而形成的无限循环。

`Worker1`和`Worker2`这些是描述如何启动各个工作进程的元组，稍后就会看到它们。

现在回到公司上来，同时构建一个监控树。

要做的第一件事是给公司选一个名字，我们决定叫它`sellaprime`。`sellaprime`监控器的工作是确保质数和面积服务器始终保持运行。为了做到这一点，我们将编写另一个用于`gen_supervisor`的回调模块。这个模块的代码如下：

sellaprime_supervisor.erl

```
-module(sellaprime_supervisor).
-behaviour(supervisor).         % 参见erl -man supervisor
-export([start/0, start_in_shell_for_testing/0, start_link/1, init/1]).
```

```
start() ->
    spawn(fun() ->
                  supervisor:start_link({local,?MODULE}, ?MODULE, _Arg = [])
          end).
start_in_shell_for_testing() ->
    {ok, Pid} = supervisor:start_link({local,?MODULE}, ?MODULE, _Arg = []),
    unlink(Pid).
start_link(Args) ->
    supervisor:start_link({local,?MODULE}, ?MODULE, Args).
init([]) ->
    %% 安装我自己的错误处理器
     gen_event:swap_handler(alarm_handler,
                                {alarm_handler, swap},
                                {my_alarm_handler, xyz}),
    {ok, {{one_for_one, 3, 10},
          [{tag1,
            {area_server, start_link, []},
            permanent,
            10000,
            worker,
            [area_server]},
           {tag2,
            {prime_server, start_link, []},
            permanent,
            10000,
            worker,
            [prime_server]}
          ]}}.
```

重点部分是`init/1`返回的数据结构。

sellaprime_supervisor.erl

```
{ok, {{one_for_one, 3, 10},
      [{tag1,
        {area_server, start_link, []},
        permanent,
        10000,
        worker,
        [area_server]},
       {tag2,
        {prime_server, start_link, []},
        permanent,
        10000,
        worker,
        [prime_server]}
      ]}}.
```

这个数据结构定义了一种监控策略。之前讨论过监控策略和重启频率，现在剩下的就是面积服务器和质数服务器的启动格式了。

Worker的格式是下面这种元组：

```
{Tag, {Mod, Func, ArgList},
       Restart,
       Shutdown,
       Type,
       [Mod1]}
```

这些参数的意义如下。

- Tag
  这是一个原子类型的标签，将来可以用它指代工作进程（如果有必要的话）。
- {Mod, Func, ArgList}
  它定义了监控器用于启动工作器的函数，将被用作apply(Mod, Fun, ArgList)的参数。
- Restart = permanent | transient | temporary
  permanent（永久）进程总是会被重启。transient（过渡）进程只有在以非正常退出值终止时才会被重启。temporary（临时）进程不会被重启。
- Shutdown
  这是关闭时间，也就是工作器终止过程允许耗费的最长时间。如果超过这个时间，工作进程就会被杀掉。（还有其他值可用，参见supervisor的手册页。）
- Type = worker | supervisor
  这是被监控进程的类型。可以用监控进程代替工作进程来构建一个由监控器组成的树。
- [Mod1]
  如果子进程是监控器或者gen_server行为的回调模块，就在这里指定回调模块名。（还有其他值可用，参见supervisor的手册页。）

这些参数看上去很吓人，但其实不是。在实践中，你可以剪切粘贴之前面积服务器代码里的值，然后插入你的模块名。这对大多数用途来说足够了。

23

23.6 启动系统

现在万事俱备，我们的公司可以开张了。好戏开始，看看有谁想买第一个质数！

首先来启动系统。

```
$ erl -boot start_sasl -config elog3
1> sellaprime_supervisor:start_in_shell_for_testing().
*** my_alarm_handler init:{xyz,{alarm_handler,[]}}
area_server starting
prime_server starting
```

现在生成一个有效查询。

```
2> area_server:area({square,10}).
100
```

监控策略是否奏效

Erlang被设计用来编写容错式系统。它最初是在瑞典电信公司爱立信的计算机科学实验室里开发的。从那时起，爱立信的OTP团队在许多内部用户的帮助下接过了开发任务。通过使用gen_server和gen_supervisor等行为，人们用Erlang构建了可靠性为99.9999999%的系统（9个9）。如果用法正确，错误处理机制能帮助你的程序永久运行（好吧，几乎永久运行）。这里介绍的错误记录器已经在线上产品里运行多年了。

现在生成一个无效查询。

```
3> area_server:area({rectangle,10,20}).
area_server stopping

=ERROR REPORT==== 15-Jul-2013::15:15:54 ===
** Generic server area_server terminating
** Last message in was {area,{rectangle,10,20}}
** When Server state == 1
** Reason for termination ==
** {function_clause,[{area_server,compute_area,[{rectangle,10,20}]},
                     {area_server,handle_call,3},
                     {gen_server,handle_msg,6},
                     {proc_lib,init_p,5}]}
  area_server starting
  ** exited: {{function_clause,
              [{area_server,compute_area,[{rectangle,10,20}]},
               {area_server,handle_call,3},
               {gen_server,handle_msg,6},
               {proc_lib,init_p,5}]},
               {gen_server,call,
                 [area_server,{area,{rectangle,10,20}}]}} **
```

不好，面积服务器崩溃了，我们触发了有意设置的错误。监控器检测到这次崩溃并重启了面积服务器。所有这些都被错误记录器记录下来，我们也看到了这个错误的打印输出。错误消息显示出问题所在：程序尝试执行area_server:compute_area({rectangle,10,20})时崩溃了，也就是错误消息function_clase的第一行所展示的。错误消息的格式是{Mod,Func,[Args]}。回去看看前几页里定义面积计算的部分（在compute_area/1里），应该能找到这个错误。

崩溃发生后，一切都恢复正常，就像构想的那样。接下来生成一个合法请求。

```
4> area_server:area({square,25}).
625
```

系统又能正常工作了。现在来生成一个小质数。

```
5> prime_server:new_prime(20).
Generating a 20 digit prime ........
37864328602551726491
```

再生成一个大质数。

```
6> prime_server:new_prime(120).
Generating a 120 digit prime
=ERROR REPORT==== 15-Jul-2013::15:22:17 ===
*** Tell the Engineer to turn on the fan
.....................................

=ERROR REPORT==== 15-Jul-2013::15:22:20 ===
*** Danger over. Turn off the fan
765525474077993399589034417231006593110007130279318737419683
288059079481951097205184294443332300308877493399942800723107
```

现在我们有了一个能正常运行的系统。如果某个服务器崩溃了，就会被自动重启，错误日志里则会有关于这个错误的信息。来看一下错误日志。

```
1> rb:start([{max,20}]).
rb: reading report...done.
rb: reading report...done.
{ok,<0.53.0>}
2> rb:list().
No                Type       Process Date       Time
==                ====       ======= ====       ====
20            progress     <0.29.0> 2013-07-30 15:05:15
19            progress     <0.22.0> 2013-07-30 15:05:15
18            progress     <0.23.0> 2013-07-30 15:05:21
17   supervisor_report     <0.23.0> 2013-07-30 15:05:21
16               error     <0.23.0> 2013-07-30 15:07:07
15               error     <0.23.0> 2013-07-30 15:07:23
14               error     <0.23.0> 2013-07-30 15:07:41
13            progress     <0.29.0> 2013-07-30 15:15:07
12            progress     <0.29.0> 2013-07-30 15:15:07
11            progress     <0.29.0> 2013-07-30 15:15:07
10            progress     <0.29.0> 2013-07-30 15:15:07
9             progress     <0.22.0> 2013-07-30 15:15:07
8             progress     <0.23.0> 2013-07-30 15:15:13
7             progress     <0.23.0> 2013-07-30 15:15:13
6                error     <0.23.0> 2013-07-30 15:15:54
5         crash_report  area_server 2013-07-30 15:15:54
4    supervisor_report     <0.23.0> 2013-07-30 15:15:54
3             progress     <0.23.0> 2013-07-30 15:15:54
2                error     <0.29.0> 2013-07-30 15:22:17
1                error     <0.29.0> 2013-07-30 15:22:20
```

有地方出问题了。我们能看到一个面积服务器的错误报告。要想查明发生了什么，可以查看错误报告。

```
9> rb:show(5).

CRASH REPORT  <0.43.0>                        2013-07-30 15:15:54
================================================================
Crashing process
pid                                                     <0.43.0>
registered_name                                      area_server
error_info
{function_clause,[{area_server,compute_area,[{rectangle,10,20}]},
                  {area_server,handle_call,3},
                     {gen_server,handle_msg,6},
                     {proc_lib,init_p,5}]}
initial_call
    {gen,init_it,
        [gen_server,
         <0.42.0>,
         <0.42.0>,
         {local,area_server},
         area_server,
         [],
         []]}
ancestors                         [sellaprime_supervisor,<0.40.0>]
messages                                                      []
links                                                 [<0.42.0>]
dictionary                                                    []
trap_exit                                                  false
status                                                   running
heap_size                                                    233
stack_size                                                    21
reductions                                                   199
ok
```

打印输出{function_clause, compute_area, ...}展示了程序里导致服务器崩溃的准确位置。定位和纠正这个错误应该是件简单的事。再看一下后面这些错误。

```
10> rb:show(2).

ERROR REPORT  <0.33.0>                        2013-07-30 15:22:17
================================================================
*** Tell the Engineer to turn on the fan
```

以及

```
10> rb:show(1).

ERROR REPORT  <0.33.0>                        2013-07-30 15:22:20
================================================================
*** Danger over. Turn off the fan
```

这些是因为计算的质数过大而引发的风扇警报！

23.7 应用程序

我们差不多已经完工了。现在要做的是编写一个扩展名为.app的文件，它包含关于这个应用程序的信息。

sellaprime.app

```
%% 这是应用程序资源文件（.app文件），
%% 它用于'base'应用程序。
{application, sellaprime,
 [{description, "The Prime Number Shop"},
  {vsn, "1.0"},
  {modules, [sellaprime_app, sellaprime_supervisor, area_server,
             prime_server, lib_lin, lib_primes, my_alarm_handler]},
  {registered,[area_server, prime_server, sellaprime_super]},
  {applications, [kernel,stdlib]},
  {mod, {sellaprime_app,[]}},
  {start_phases, []}
 ]}.
```

现在必须编写一个回调模块，它的名称与前面文件里的mod文件名相同。这个文件取自A.3节里的模板，然后加以填写。

sellaprime_app.erl

```
-module(sellaprime_app).
-behaviour(application).
-export([start/2, stop/1]).
start(_Type, StartArgs) ->
    sellaprime_supervisor:start_link(StartArgs).
stop(_State) ->
    ok.
```


它必须导出函数start/2和stop/1。做完这一切之后，就可以在shell里启动和停止应用程序了。

```
$ erl -boot start_sasl -config elog3
1> application:loaded_applications().
[{kernel,"ERTS  CXC 138 10","2.16.1"},
 {sasl,"SASL  CXC 138 11","2.3.1"},
 {stdlib,"ERTS  CXC 138 10","1.19.1"}]
2> application:load(sellaprime).
ok
3> application:loaded_applications().
[{sellaprime,"The Prime Number Shop","1.0"},
 {kernel,"ERTS  CXC 138 10","2.16.1"},
 {sasl,"SASL  CXC 138 11","2.3.1"},
 {stdlib,"ERTS  CXC 138 10","1.19.1"}]
4> application:start(sellaprime).
*** my_alarm_handler init:{xyz,{alarm_handler,[]}}
```

```
area_server starting
prime_server starting
ok
5> application:stop(sellaprime).
prime_server stopping
area_server stopping

=INFO REPORT==== 26-May-2013::14:16:57 ===
application: sellaprime
exited: stopped
type: temporary
ok
6> application:unload(sellaprime).
ok
7> application:loaded_applications().
[{kernel,"ERTS  CXC 138 10","2.16.1"},
{sasl,"SASL  CXC 138 11","2.3.1"},
{stdlib,"ERTS  CXC 138 10","1.19.1"}]
```

现在它就是一个功能完备的OTP应用程序了。我们在第2行里载入了应用程序，这么做会载入全部代码，但不会启动应用程序。第4行启动了应用程序，第5行则停止了它。请注意，启动和停止应用程序时我们能看到打印输出，因为它调用了面积服务器和质数服务器里相应的回调函数。在第6行卸载了应用程序，这样应用程序里的所有模块代码都被移除了。

用OTP构建复杂的系统时，会把它们打包成应用程序。这样我们就能统一启动、停止和管理它们。

请注意，用`init:stop()`关闭系统时，所有运行中的应用程序会按顺序一一关闭。

```
$ erl -boot start_sasl -config elog3
1>  application:start(sellaprime).
*** my_alarm_handler init:{xyz,{alarm_handler,[]}}
area_server starting
prime_server starting
ok
2> init:stop().
ok
prime_server stopping
area_server stopping
$
```

命令2后面的两行文本来自面积与质数服务器，这就表示各个`gen_server`回调模块里的`terminate/2`函数都得到了调用。

23.8　文件系统组织方式

我还没有提到过文件系统的组织方式，这是有原因的，我的目的是每次只解决一个问题。

规范的应用程序各个部分所属的文件通常都处于定义明确的位置。这不是必要条件，只要相

关文件能在运行时找到，文件是如何组织的并不重要。

我把本书的大多数演示文件都放在同一个目录里。这么做简化了示例，也避免了搜索路径和不同程序之间的交互等问题。

sellaprime公司使用的主要文件如下。

文 件	内 容
area_server.erl	面积服务器（gen_server回调模块）
prime_server.erl	质数服务器（gen_server回调模块）
sellaprime_supervisor.erl	监控器回调模块
sellaprime_app.erl	应用程序回调模块
my_alam_handler.erl	用于gen_event的事件回调模块
sellaprime.app	应用程序规范
elog4.config	错误记录器配置文件

要了解这些文件和模块是如何使用的，可以来看看启动应用程序时发生的事件序列。

(1) 用下列命令启动系统：

```
$ erl -boot start_sasl -config elog4.config
1> application:start(sellaprime).
...
```

sellaprime.app文件必须位于Erlang的启动根目录或它的子目录里。

应用程序控制器随后在sellaprime.app里寻找一个{mod, ...}声明。它包含应用程序控制器的名称，在这个案例里是模块sellaprime_app。

(2) 回调方法sellaprime_app:start/2被调用。

(3) sellaprime_app:start/2 调用 sellaprime_supervisor:start_link/2，启动sellaprime监控器。

(4) 监控器回调函数sellaprime_supervisor:init/1被调用，它会安装一个错误处理器，然后返回一个监控规范。这个监控规范说明了如何启动面积服务器和质数服务器。

(5) sellaprime监控器启动面积服务器和质数服务器，两者都是gen_server的回调模块。

停止这一切很容易，只需要调用application:stop(sellaprime)或init:stop()。

23.9 应用程序监视器

应用程序监视器是一个用来查看应用程序的GUI。appmon:start()命令会启动应用程序查看器。输入这个命令后，你会看到一个如图23-2所示的窗口，必须点击一个应用程序才能进行查看。图23-3展示了应用程序sellaprime在应用程序监视器里的样子。

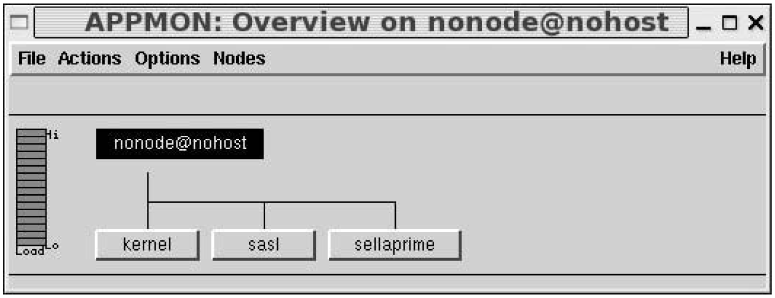


图23-2 应用程序监视器初始窗口

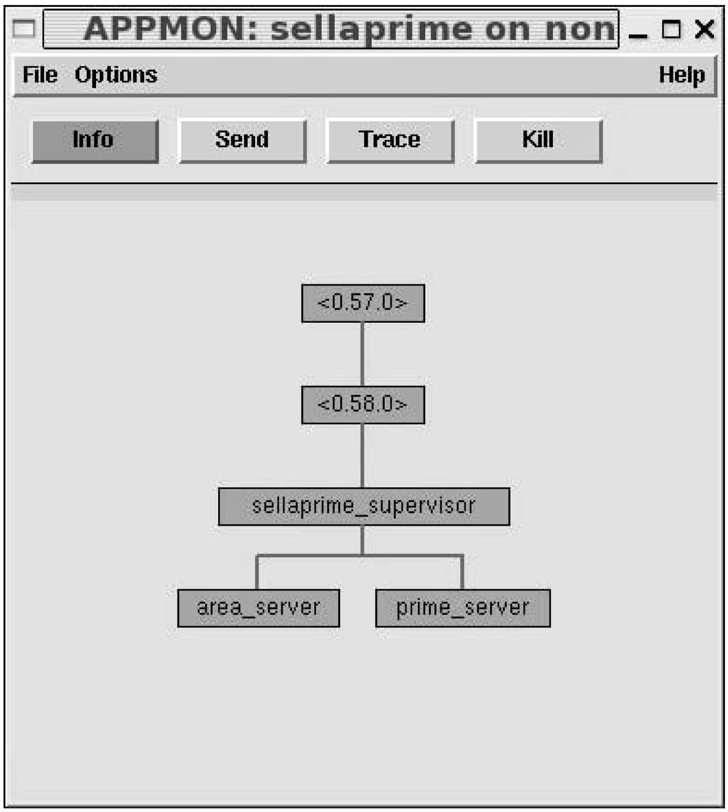


图23-3 sellaprime应用程序

23.10 怎样计算质数

计算质数很简单。代码如下：

lib_primes.erl

```
-module(lib_primes).
-export([make_prime/1, is_prime/1, make_random_int/1]).

make_prime(1) ->
    lists:nth(random:uniform(4), [2,3,5,7]);
make_prime(K) when K > 0 ->
    new_seed(),
    N = make_random_int(K),
    if N > 3 ->
            io:format("Generating a ~w digit prime ",[K]),
            MaxTries = N - 3,
            P1 = make_prime(MaxTries, N+1),
            io:format("~n",[]),
            P1;
        true ->
            make_prime(K)
    end.

make_prime(0, _) ->
    exit(impossible);
make_prime(K, P) ->
    io:format(".",[]),
    case is_prime(P) of
        true  -> P;
        false -> make_prime(K-1, P+1)
    end.

is_prime(D) when D < 10 ->
    lists:member(D, [2,3,5,7]);
is_prime(D) ->
    new_seed(),
    is_prime(D, 100).

is_prime(D, Ntests) ->
    N = length(integer_to_list(D)) -1,
    is_prime(Ntests, D, N).

is_prime(0, _, _) -> true;
is_prime(Ntest, N, Len) ->
    K = random:uniform(Len),
    %% A是一个小于K的随机数
    A = make_random_int(K),
    if
        A < N ->
            case lib_lin:pow(A,N,N) of
                A -> is_prime(Ntest-1,N,Len);
                _ -> false
            end;
```

```
-          true ->
50             is_prime(Ntest, N, Len)
-      end.
-
- %% make_random_int(N) -> 一个N位的随机整数
- make_random_int(N) -> new_seed(), make_random_int(N, 0).
55
- make_random_int(0, D) -> D;
- make_random_int(N, D) ->
-     make_random_int(N-1, D*10 + (random:uniform(10)-1)).
```

make_prime(K)返回一个至少K位的质数。为了做到这一点，我们使用一种基于Bertrand假说的算法。Bertrand假说是指对任意自然数N > 3，都有一个质数P满足N < P < 2N - 2。Tchebychef在1850年证明了它，Erdos在1932年对证明做了改进。

首先生成一个N位的随机整数（第54~58行），然后测试N+1和N+2等数字是否是质数。这是在第19~26行的循环里完成的。

is_prime(D)会在D很可能是质数时返回true，否则返回false。它运用了费马小定理：如果N是质数且A < N，那么A^N mod N = A。因此，为了测试N是否是质数，我们生成小于N的随机值A并运行费马测试。如果测试失败，N就不是一个质数。这是一种概率性测试，所以每运行一次测试，N是质数的概率就会增加。测试是在第38~51行里执行的。

是时候生成一些质数了。

```
1>  lib_primes:make_prime(500).
Generating a 500 digit prime ..............................
7910157269872010279090555971150961269085929213425082972662439
1259263140285528346132439701330792477109478603094497394696440
4399696758714374940531222422946966707622926139385002096578309
0625341667806032610122260234591813255557640283069288441151813
9110780200755706674647603551510515401742126738236731494195650
5578474497545252666718280976890401503018406521440650857349061
2139806789380943526673726726919066931697831336181114236228904
0186804287219807454619374005377766827105603689283818173007034
056505784153
```

我们已经概述了构建OTP应用程序的基础知识。OTP应用程序有标准化的文件格式，启动和停止的方式也很规则。它们通常包括一个被gen_supervisor监控的gen_server，外加一些错误记录代码。可以在Erlang分发套装的网站上找到所有关于OTP行为的完整介绍。

23.11　深入探索

我在这里省略了很多细节，只解释了相关的原则。你可以在gen_event、error_logger、supervisor和application的手册页里找到详细介绍。

更多OTP设计原则的细节可以在Erlang/OTP系统文档[①]里找到。

① http://www.erlang.org/doc/pdf/otp-system-documentation.pdf

在目前这个阶段，我们已经涉及了构建兼容OTP框架的常规Erlang应用程序所需要的全部重要主题。

在本书最后一部分，我们将跳出OTP框架，向你展示一些额外的编程技巧并构建复杂的示例程序。

23.12 练习

浏览这些练习时请别担心。当你到达清单最后的问题时，将会面临一个难度很高的问题。要解决它，你需要了解OTP行为（本章和上一章），理解如何使用Mnesia（用于数据复制），还要对分布式Erlang有基本的了解（如何设立连接节点）。

这些主题单独看来并不算特别复杂。若把简单的事物组合到一起，最终表现出的行为就可能会非常复杂。即使你不去立即解决这些问题，单单思考它们也能帮助你把这类问题的解决方案分解成可驾驭的几个部分。要构建大型的容错式系统，我们需要考虑服务器该做什么，如何重启崩溃的服务器，如何进行负载均衡，如何/向何处复制数据等问题。这些练习的排列顺序能指引你逐步完成这个过程。

(1) 制作一个名为`prime_tester_server`的`gen_server`，让它测试给定的数字是否是质数。你可以使用`lib_primes.erl`里的`is_prime/2`函数来处理（或者自己实现一个更好的质数测试函数）。把它添加到`sellaprime_supervisor.erl`的监控树里。

(2) 制作由10个质数测试服务器组成的进程池。制作一个队列服务器来把请求加入队列，直到其中一个质数测试服务器处于空闲状态为止。当质数测试服务器空闲时，向它发送一个请求来测试某个数字是否是质数。

(3) 修改质数测试服务器的代码，让它们各自维护一个请求队列，然后移除队列服务器。编写一个负载均衡器来记录各个质数测试服务器中正在进行的任务和待完成请求。测试新质数的请求现在应该发送到负载均衡器。安排负载均衡器把请求发送给负载最小的服务器。

(4) 实现一种监控层级体系，使任何质数测试服务器崩溃后都能被重启。如果负载均衡器崩溃了，就让所有质数测试服务器都崩溃，然后全体重启。

(5) 使全体重启所需的数据在两台机器上同步复制。

(6) 实现一种重启策略，使整台机器崩溃后也能全体重启。

Part 5 第五部分

构建应用程序

在这一部分里，我们将会看到一些编写 Erlang 程序时常用的编程术语，还会了解如何把第三方代码集成到我们的应用程序里，相比由自己完成所有工作，这种方式能更快得到结果。我们还将学习如何让程序在多核计算机上并行。最后，将解决福尔摩斯的最后一案。

第 24 章

编程术语

本章将探究一些编程术语并介绍Erlang代码的不同组织方式。我们将从一个示例入手，展示应该如何看待编程世界以及在这个世界里所见到的对象。

24.1 保持 Erlang 世界观

Erlang的世界观是一切都是进程和进程只能通过交换消息进行互动。这种世界观让我们的设计具备了概念完整性，也更易于理解。

假设想用Erlang编写一个Web服务器。有一位用户向我们的Web服务器请求一个名为`hello.html`的网页。最简单的Web服务器如下：

```
web_server(Client) ->
   receive
        {Client, {get, Page}} ->
            case file:read(Page) of
                {ok, Bin} ->
                    Client ! {self(), {data, Bin}};
                {error, _} ->
                    Client ! {self(), error}
            end,
            web_server(Client)
    end.
```

但这段代码如此简单是因为它只接收和发送Erlang数据类型，客户端发送的可不是Erlang数据类型，而是复杂程度远超前者的HTTP请求。HTTP请求建立在TCP连接上，请求自身也可能会分段发送，所有这一切都使得服务器程序的复杂程度远超之前所展示的简单代码。

为了让事情更简单，我们在Erlang服务器和接收HTTP客户端消息的TCP驱动之间插入一个名为中间人的进程。中间人会解析HTTP请求，并把它们转变成Erlang消息，如图24-1所示。可以看出翻译进程为何被称作中间人，它就位于TCP驱动和Web服务器中间。

对服务器来说，外部世界里的对象只会“说”Erlang语言。相比用一个进程做两件事（处理HTTP请求和服务请求），我们现在有了两个角色定义明确的进程。中间人只知道如何转换HTTP消息和Erlang消息，而服务器对HTTP协议的细节一无所知，只负责处理纯Erlang消息。把进程一分为二不仅让设计更清晰，还有一个额外的好处：它可以增加并发性，这两个进程可以并行执行。



图　24-1

图24-2展示了与处理HTTP请求有关的消息流。

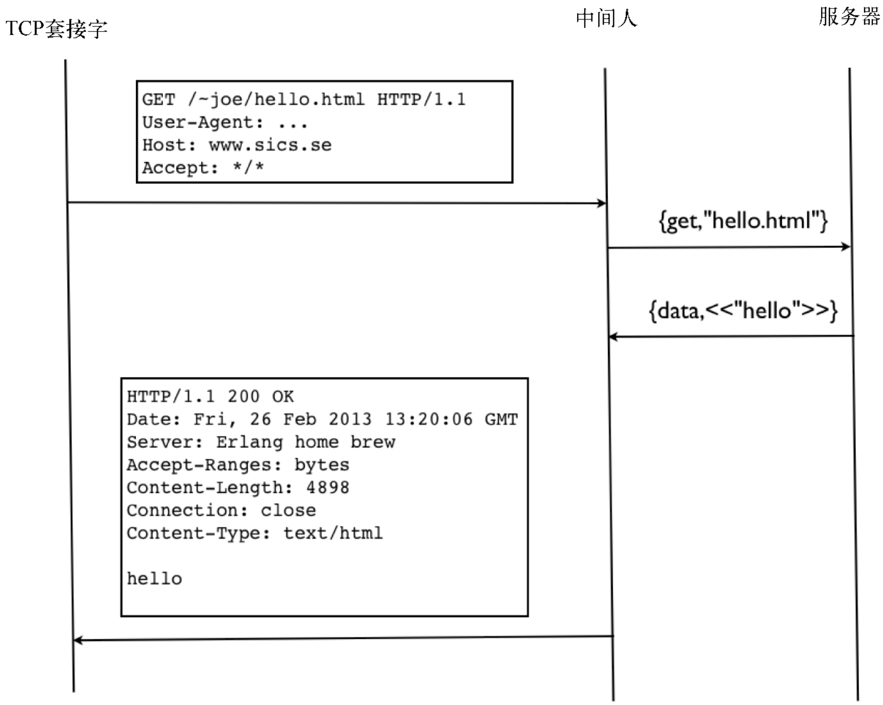


图24-2　Web服务器协议

中间人的确切工作原理不属于我们的讨论范围。它要做的就是解析传入的HTTP请求，把它们转换成Erlang数据类型，并把传出的Erlang数据类型转换成HTTP响应。

在这个例子里，我们选择抽象掉HTML请求的许多细节。HTML请求头包含了许多没有在此展示的额外信息。作为中间人设计的一部分，必须决定要对Erlang应用程序暴露多少底层协议的细节。

假设想要扩展这个例子，让它能响应FTP文件请求或通过IRC信道发送的文件。可以向图24-3所展示的那样组织系统里的进程。

HTTP、FTP和IRC使用完全不同的协议进行机器间的文件传输。事实上，IRC不支持文件传

输，文件传输通常由直接端对端（Direct Client to Client，简称CDC）协议实现，而大多数IRC客户端都支持这个协议。

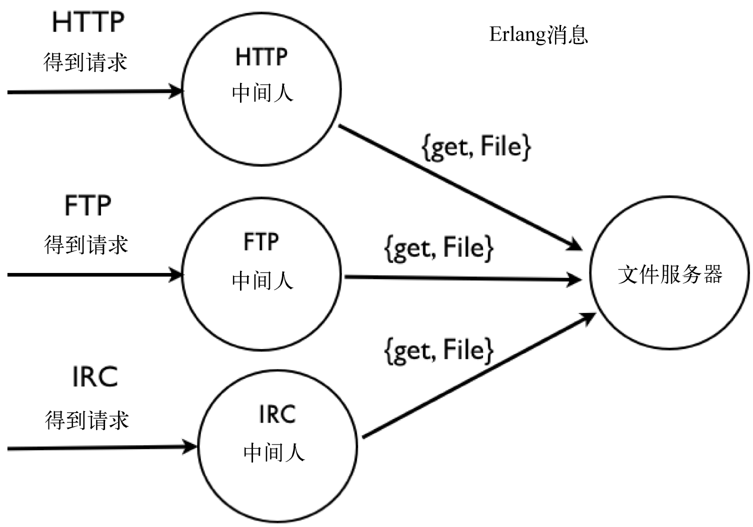


图24-3　消息统一化

当中间人把这些外部协议转换成Erlang消息后，就可以把单个Erlang服务器用作所有不同协议的后端了。

统一化的Erlang消息确实简化了实现工作，它有以下一些优点。

- 它抽象掉了不同线路协议（比如HTTP和FTP协议）之间的区别。
- Erlang消息无需解析，接收进程不必先解析消息再处理它。相比之下，HTTP服务器就必须解析接收到的所有消息。
- Erlang消息可以包含任意复杂度的数据类型。相比之下，HTTP消息必须被序列化成扁平的形式才能传输。
- Erlang消息可以在处理器之间传送，或者以一种简单通用的序列化格式保存在数据库里。

24.2　多用途服务器

摒弃了各种服务需要不同的消息格式这一概念后，就可以用消息统一化来解决多种问题了。例如，这里有一个“多用途服务器”：

multi_server.erl

```
-module(multi_server).
-export([start/0]).

```

```
4  start() -> spawn(fun() -> multi_server() end).
5
6  multi_server() ->
7      receive
8          {_Pid, {email, _From, _Subject, _Text} = Email} ->
9              {ok, S} = file:open("mbox", [write,append]),
10             io:format(S, "~p.~n", [Email]),
11             file:close(S);
12         {_Pid, {im, From, Text}} ->
13             io:format("Msg (~s): ~s~n",[From, Text]);
14         {Pid, {get, File}} ->
15             Pid ! {self(), file:read_file(File)};
16         Any ->
17             io:format("multi server got:~p~n",[Any])
18     end,
19     multi_server().
```

这段代码模拟了许多常见服务的关键行为。

- 在第8~11行里，它表现出电子邮件客户端的行为。

  电子邮件客户端的关键任务是接收邮件并把它保存到你的计算机里（按惯例是一个名为mbox的文件）。我们接收一个消息，打开名为mbox的文件，把消息写入这个文件，任务完成。
- 在第12~13行里，它表现出即时通讯客户端的行为。

  即时通讯客户端的关键任务是接收一个消息并告知用户。通过向控制台写入消息来告知用户。
- 在第14~15行里，它表现出FTP/RCP/HTTP服务器的行为。

  FTP服务器、HTTP服务器或其他任何文件服务器的关键任务是把一个文件从服务器传送到客户端。

看了这段代码，我们意识到其实并不需要那么多各不相同的客户端–服务器请求和响应编码，有一种通用的格式就足够了。一切消息都使用Erlang数据类型。

所有这些好东西都能在分布式环境里工作，这要归功于两个内置函数：term_to_binary(Term)和逆函数binary_to_term(Bin)，后者用于恢复数据类型。

在分布式系统里，binary_to_term(Bin)可以根据Bin里保存的数据类型外部表现形式来重建任何数据类型。Bin一般通过套接字进入机器，不过具体的细节在此并不重要。binary_to_term只是简单地重建这个数据类型，而在像HTTP这样的协议里，输入的请求必须进行解析，这就导致了整个过程效率低下。

可以添加一对执行规定任务的对称函数来实现加密层和压缩层。这里有一个例子：

```
send1(Term) -> encrypt(compress(term_to_binary(Term))).

receive1(Bin) -> binary_to_term(decompress(decrypt(Bin))).
```

如果想要通过网络发送加密并压缩过的移动代码[①]，就可以这么写：

```
send_code(Mod, Func, Args) ->
    encrypt(compress(term_to_binary({Mod,Func,Args}))).

receive_code(Bin) ->
   {Mod, Func, Args} = binary_to_term(decompress(decrypt(Bin))),
   apply(Mod, Func, Args).
```

在这里组合了三个概念：用`term_to_binary`和它的逆函数通过网络发送数据类型，用`apply`执行代码，以及用对称函数组来压缩/解压缩和加密/解密数据。请注意，这些压缩/解压缩和加密/解密函数不是Erlang的内置函数，我们只是假设存在这样的函数。

对Erlang程序员而言，这个世界很美好。编写了恰当的中间人进程之后，所有外部进程就都会说Erlang语言了。这让复杂系统真正得到简化，特别是在使用多种不同外部协议的情况下。

这就像是一个人人都说英语（或普通话）的世界——交流变得容易多了。

24.3　有状态的模块

通过使用一种被称为*元组模块*的机制，可以把状态和模块名封装到一起。可以用这种机制来隐藏信息和创建适配器模块，后者会对使用接口的程序隐藏接口细节。如果想制作面向多个不同模块的接口，或者模拟面向对象编程的某些特性，这种机制就非常有用。

调用`X:Func(....)`时，`X`不必非得是一个原子，它还可以是元组。如果先写`X = {Mod, P1, P2, ..., Pn}`然后调用`X:Func(A1, A2, ..., An)`，那么实际调用的是`Mod:Func(A1, A2, ..., An, X)`。举个例子，`{foo,1,2,3}:bar(a,b)`这个调用会被转换成`foo:bar(a,b,{foo,1,2,3})`。

可以用这种机制来创建“有状态的”模块。首先我们将用一个简单的有状态计数器进行演示，然后再转向一个示例，它将为两个现有模块创建一个适配器模块。

有状态的计数器

为了演示元组模块这个概念，我们将从一个简单的计数器示例入手。它带有一个状态参数`N`，用来表示计数器的值。它的代码如下：

counter.erl

```
-module(counter).
-export([bump/2, read/1]).

bump(N, {counter,K}) ->  {counter, N + K}.
read({counter, N}) -> N.
```

可以像下面这样来测试这段代码。首先来编译模块。

① 移动代码（mobile code）是指在系统间传输且无需安装就能执行的代码。——译者注

```
1> c(counter).
{ok,counter}
```

然后创建一个元组模块的实例。

```
2> C = {counter,2}.
{counter, 2}
```

然后调用get/0。

```
3> C:read().
2
```

因为C是一个元组，所以它会被转换成counter:read({counter,2})，这个调用的返回值是2。

```
3> C1 = C:bump(3).
{counter, 5}
```

C:bump(3)被转换成counter:bump(3, {counter, 2})，因此返回{counter, 5}。

```
4> C1:read().
5
```

值得注意的是，对元组C和C1的调用代码来说，模块名counter和状态变量都是隐藏的。

24.4 适配器变量

假设有两个或多个功能相近的库,但无法决定要用哪一个。这些库可能具有相似的函数接口，但性能特征不同。以键–值存储为例，第一种存储把键和值放在内存里，第二种把键放在内存里，把值放在硬盘上。也许还有第三种——把值较小的键放在内存里，值较大的则放在硬盘上。即使是键–值存储这么简单的事情，也可能存在多种不同的存储实现方式。

假设想编写一些利用键–值存储的代码。编写这个应用程序时，必须做一个设计决策——从这些可用选项里选择一种键–值存储。也许过了很久之后，某些设计决策被证明是错误的，那时我们也许会想要改变后端的存储方式。但是，如果用来访问新旧存储方式的API不一致，就必须对程序做大量的修改。

这就是适配器模式大显身手的时候了。适配器是一种元组模块，它能为应用程序提供一组统一的接口。

我们将构建一个适配器模式来演示这一点，这个模式会为用lists和dict模块实现的键-值存储提供统一的接口。这个适配器的接口如下。

- adapter_db1:new(Type :: dict | lists) -> Mod

  创建一个Type类型的新键–值存储。它会返回一个元组模块Mod。

- Mod:store(Key, Val) -> Mod1

  存储一个Key, Value对。Mod是旧的存储状态，Mod1是新的存储状态。

24

❑ Mod:lookup(Key) -> {ok, Val} | error
在存储里查找Key键。如果能找到值就返回{ok, Val}，否则返回error。

可以编写如下代码来使用这组API：

```
M0 = adapter_db1:new(dict), ...
M1 = M0:store(Key1, Val2),
M2 = M1:store(Key2, Val2),
...
ValK = MK:lookup(KeyK),
```

如果想用lists实现，就可以把创建模块的代码行改成Mod = adapter_db1:new(lists)。出于兴趣，可以把它与用于dict模块的编程样式进行对比。为dict编写的代码如下：

```
D0 = dict:new(),
D1 = dict:store(Key1, Val1, D0),
D2 = dict:store(Key2, Val2, D1),
...
ValK = dict:find(KeyK, Dk)
```

用来访问元组模块的代码要略短一些，因为可以把所有内部细节都隐藏在一个Modi变量里。使用dict需要两个参数：变量名和字典结构本身。

现在来编写适配器。

adapter_db1.erl

```
-module(adapter_db1).
-export([new/1, store/3, lookup/2]).

new(dict) ->
    {?MODULE, dict, dict:new()};
new(lists) ->
    {?MODULE, list, []}.

store(Key, Val, {_, dict, D}) ->
    D1 = dict:store(Key, Val, D),
    {?MODULE, dict, D1};
store(Key, Val, {_, list, L}) ->
    L1 = lists:keystore(Key, 1, L, {Key,Val}),
    {?MODULE, list, L1}.

lookup(Key, {_,dict,D}) ->
    dict:find(Key, D);
lookup(Key, {_,list,L}) ->
    case lists:keysearch(Key, 1, L) of
        {value, {Key,Val}} -> {ok, Val};
        false              -> error
    end.
```

这一次我们的模块是用一个{adapter_db1, Type, Val}形式的元组来表示的。如果Type

是list，Val就是一个列表；如果Type是dict，Val则是一个字典。

可以在一个单独模块里编写一些简单的代码来测试这个适配器。

adapter_db1_test.erl

```
-module(adapter_db1_test).
-export([test/0]).
-import(adapter_db1, [new/1, store/2, lookup/1]).
test() ->
    %% 测试dict模块
    M0 = new(dict),
    M1 = M0:store(key1, val1),
    M2 = M1:store(key2, val2),
    {ok, val1} = M2:lookup(key1),
    {ok, val2} = M2:lookup(key2),
    error = M2:lookup(nokey),
    %% 测试lists模块
    N0 = new(lists),
    N1 = N0:store(key1, val1),
    N2 = N1:store(key2, val2),
    {ok, val1} = N2:lookup(key1),
    {ok, val2} = N2:lookup(key2),
    error = N2:lookup(nokey),
    ok.
```

```
1> adapter_db1_test:test().
ok
```

测试成功。这样就实现了目标，也就是把两个接口不同的模块隐藏在一个适配器模块之后，让它提供对这两个模块的公共接口。

适配器适合为已经存在的代码提供通用接口。适配器的接口可以保持不变，而它背后的代码可以被修改来反映出不同的需求。

24.5 表意编程

作为一种编程风格，表意编程（intentional programming）能让我们轻易看出程序员的意图。相关函数的名称应当能明显体现出程序员的意图，而不是要通过分析代码的结构才能推断出来。

一个例子胜过千言万语。在早期的Erlang里，库模块dict导出了一个lookup/2函数，它的接口如下：

```
lookup(Key, Dict) -> {ok, Value} | not_found
```

根据这个定义，lookup可以用在三种上下文环境里。

(1) 要检索数据，可以编写如下代码：

```
{ok, Value} = lookup(Key, Dict)
```

此处的lookup被用来从字典里提取带有已知键的项。如果字典里不存在这个键，就会返回not_found，从而导致模式匹配错误，程序会抛出一个异常错误。退出的原因是{badmatch, not_found}，这个错误消息有待改进，把错误原因换成{bad_key, Key}的话信息量就会更丰富。

(2) 要进行搜索，可以编写如下代码：

```
case lookup(Key, Dict) of
   {ok, Val} ->
      ... 对Val进行操作 ...
   not_found ->
      ... 做另一些事 ...
end.
```

可以看出程序员不知道字典里是否有这个键，因为他们编写的代码模式匹配了lookup的所有返回值（{ok, Val}和not_found）。我们还能看出程序对Val进行了某种操作（根据注释）。从这段代码可以推断程序员是想在字典里搜索某个值。当不知道某个东西的位置时，就会进行搜索。

(3) 要测试某个键是否存在，下面的代码片段：

```
case lookup(Key, Dict) of
    {ok, _} ->
       ... 做一些事 ...
    not_found ->
       ... 做另一些事 ...
end.
```

会测试字典里是否存在特定的键。得出这个推论是因为注意到lookup的两种返回值都有模式匹配，但找到的项目值却从未被使用，能看出这一点是因为模式匹配的是{ok, _}而不是{ok, Val}（就像第一个例子那样）。既然这个键的关联值未被使用，就可以假定调用lookup是为了测试某个键是否存在。

前面三个例子让lookup函数出现了含义超载。它有三种不同的用途：数据检索、搜索和测试某个键是否存在。

与其猜测程序员的意图和分析代码，不如调用一个能显式表达三种意图之一的库方法。dict为此导出了三个函数。

```
dict:fetch(Key, Dict) = Val | EXIT
dict:search(Key, Dict) = {found, Val} | not_found.
dict:is_key(Key, Dict) = Boolean
```

这些函数能准确表达程序员的意图。无需猜测和分析程序，函数名就能清楚表达程序员的意图。如果字典里可能存在某个键，就调用search，但不存在也算不上错误。如果字典里必定存在某个键，就调用fetch，此时若不存在就算是错误。要测试字典里是否存在某个键则用is_key。

用lookup编写的代码会比用fetch、search和is_key三者之一编写的代码更难以理解和维护。

本章最重要的是中间人这个概念。另一个极其重要的概念是外部世界里的一切都应该建模为Erlang进程，它是让组件无缝结合的核心秘诀。

在下一章里，我们将了解如何共享代码以及如何集成自己与他人的工作成果。另外，还将看一些在本书某些示例中用过的第三方工具。借用他人的代码能让我们更快解决问题，也可以通过分享自己的代码来帮助他人。如果你帮助别人，别人也会帮助你。

24.6 练习

(1) 扩展`adapter_db1`里的适配器，使调用`adapter_db1:new(persistent)`能创建一个持久性数据存储的元组模块。

(2) 编写一种键–值存储，让它把较小的值放入内存，把较大的值放入磁盘。制作一个适配器模块来实现它，并让这个模块与本章前面的适配器具有相同的接口。

(3) 编写一种键–值存储，让它把各个键–值对分为易失性存储和非易失性存储。调用`put(Key, memory, Val)`会把一个`Key, Val`对放入内存，`put(Key, disk, Val)`则会把数据存入磁盘。用一对进程来做这件事，一个用于易失性存储，另一个用于非易失性存储。重用本章前面的代码。

第 25 章 第三方程序

本章将讨论第三方程序，也就是由用户编写和分发的Erlang程序。这类程序的首要来源是GitHub。在这一章里，我们将看到三个来源于GitHub的流行程序。还将看到如何创建和宣传一个新的GitHub项目，以及如何把某个GitHub项目包含在自己的应用程序里。我们将会了解下面这些程序。

- rebar：rebar由Dave Smith编写，它已经成为管理Erlang项目的事实标准。通过使用rebar，用户可以创建新项目、编译项目、打包它们，以及把它们与其他项目整合在一起。rebar集成了GitHub，这样用户就能轻松获取其他来自GitHub的rebar项目，并让自己的应用程序整合这些项目。
- bitcask：bitcask由Basho公司[1]①的人编写，它是一种持久性的键–值磁盘存储，速度很快，而且“不怕崩溃”，意思是它能在崩溃重启后快速恢复。
- cowboy：cowboy由Loïc Hoguin编写，它是一个用Erlang编写的高性能Web服务器，正在成为嵌入式Web服务器的热门实现方式。我们曾把cowboy服务器用于第18章里的代码。

25.1 制作可共享代码存档并用 rebar 管理代码

在这一节里，我们将一步步制作一个开源Erlang项目，并把它托管在GitHub上。我会假定你已经有一个GitHub账号。我们将用rebar来管理这个项目。

我们会做下面这些事。

(1) 安装rebar。

(2) 在GitHub上创建一个新项目。

(3) 在本地克隆这个项目。

(4) 用rebar添加项目样板代码。

(5) 用rebar编译我们的项目。

(6) 将我们的项目上传到GitHub。

25.1.1 安装rebar

rebar可以在https://github.com/basho/rebar里找到。你应该能在https://github.com/rebar/rebar/

① http://basho.com

wiki/rebar找到rebar的预编译二进制文件。要安装rebar，请复制该文件，修改文件模式为可执行，然后把它放在路径里的某个地方。

做完这些之后，应当测试一下是否能运行rebar。

```
$ rebar -V
rebar 2.0.0 R14B04 20120604_145614 git 0f24d93
```

25.1.2 在GitHub上创建一个新项目

假设要制作一个名为bertie的新项目（我经常用Alexander McCall Smith书里的人物命名我的项目）。第一步是创建一个新的GitHub项目，为此我登录GitHub账户并按照说明来创建一个新的储存库（repository）。

(1) 点击登录页工具条右上方的"Create a new repo"（创建一个新储存库）图标，它看上去就像一本带有加号的书。

(2) 把它设置成带自述文件的公开储存库。

(3) 然后点击"Create Repository"（创建存储库）。

25.1.3 在本地克隆这个项目

我家中的机器里有一个名为`${HOME}/published`的目录，我用它存放所有的共享项目。我移至这个`published`目录，然后克隆GitHub储存库。

```
$ cd ~/published
$ git clone git@github.com:joearms/bertie.git
Cloning into 'bertie'...
Identity added: /Users/joe/.ssh/id_rsa (/Users/joe/.ssh/id_rsa)
remote: Counting objects: 3, done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 3 (delta 0), reused 0 (delta 0)
Receiving objects: 100% (3/3), done.
```

现在我通常会检查一下是否能把改动写入储存库。因此，我修改了自述文件并把它推回储存库。

```
$ emacs -nw  README.md
$ git add README.md
$ git commit README.md
$ git push
Counting objects: 5, done.
Delta compression using up to 2 threads.
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 326 bytes, done.
Total 3 (delta 0), reused 0 (delta 0)
To git@github.com:joearms/bertie.git
6b9b6b9..6b0148e  master -> master
```

看到这个结果，我松了一口气，不禁赞叹现代科技的神奇。

现在我有了一个本地目录~/published/bertie，它与GitHub储存库git@github.com:joearms/bertie.git是同步的。

25.1.4　制作一个OTP应用程序

现在移至bertie目录，然后用rebar创建一个标准OTP应用程序。

```
$ cd ~/published/bertie
$ rebar create-app appid=bertie
==> bertie (create-app)
Writing src/bertie.app.src
Writing src/bertie_app.erl
Writing src/bertie_sup.erl
```

rebar create-app命令创建了标准OTP应用程序所需的样板文件和目录结构。

现在给~/published/bertie/src目录添加一个bertie.erl模块。

```
-module(bertie).
-export([start/0]).

start() -> io:format("Hello my name is Bertie~n").
```

然后用rebar来编译这一切。

```
> rebar compile
==> bertie (compile)
Compiled src/bertie.erl
Compiled src/bertie_app.erl
Compiled src/bertie_sup.erl
```

现在我们就得到了一个完整的程序，剩下的事就是把它推回储存库。

```
$ git add src
$ git commit
$ git push
```

25.1.5　宣传你的项目

现在你已经编写代码并把它发布到GitHub上了，下一步就是宣传它。最显而易见的方式就是在Erlang邮件列表[①]上发一条简短的通告，或者发一条带#erlang标签的推特。

如果有用户想要使用你的应用程序，他们要做的就是下载它并运行rebar compile来构建这个应用程序。

① http://erlang.org/mailman/listinfo/erlang-questions

25.2 整合外部程序与我们的代码

我们已经演示了在GitHub上发布你的劳动成果所需的步骤，接下来看看如何让我们的项目包含他人的劳动成果。下面这个例子将把来自bitcask的代码整合到bertie项目中。

修改bertie，让它在启动时打印出已启动次数。例如，当bertie第10次启动时，让它进行如下通报：

```
Bertie has been run 10 times
```

为了做到这一点，我们将把bertie的已运行次数保存在一个bitcask数据库里。在bitcask里，键和值都必须是二进制型。选择二进制型<<"n">>作为键，term_to_binary(N)作为值，其中N是bertie的已运行次数。现在bertie.erl的代码如下：

bertie/bertie.erl

```
-module(bertie).
-export([start/0]).

start() ->
    Handle = bitcask:open("bertie_database", [read_write]),
    N = fetch(Handle),
    store(Handle, N+1),
    io:format("Bertie has been run ~p times~n",[N]),
    bitcask:close(Handle),
    init:stop().

store(Handle, N) ->
    bitcask:put(Handle, <<"bertie_executions">>, term_to_binary(N)).
fetch(Handle) ->
    case bitcask:get(Handle, <<"bertie_executions">>) of
        not_found -> 1;
        {ok, Bin} -> binary_to_term(Bin)
    end.
```

为了让bertie应用程序包含bitcask，将创建一个名为rebar.config的“依赖项”文件，然后把它保存在bertie项目的顶级目录里。rebar.config的代码如下：

25

bertie/rebar.config

```
{deps, [
  {bitcask, ".*", {git, "git://github.com/basho/bitcask.git", "master"}}
]}.
```

我还添加了一个makefile。

bertie/Makefile

```
all:
	test -d deps || rebar get-deps
```

```
        rebar compile
        @erl -noshell -pa './deps/bitcask/ebin' -pa './ebin' -s bertie start
```

当第一次运行这个makefile时，会看到以下输出：

```
$ ejoearm@ejoearm-eld:~/published/bertie$ make
==> bertie (get-deps)
Pulling bitcask from {git,"git://github.com/basho/bitcask.git","master"}
Cloning into 'bitcask'...
==> bitcask (get-deps)
Pulling meck from {git,"git://github.com/eproxus/meck"}
Cloning into 'meck'...
==> meck (get-deps)
rebar compile
==> meck (compile)
...
Bertie has been run 1 times
```

rebar get-deps命令从GitHub获取bitcask并把它保存在名为deps的子目录里。bitcask自身需要一个名为meck的程序用于测试，这就是所谓的递归依赖。rebar会递归获取bitcask所需的各个依赖项，并把它们保存在deps子目录里。

makefile给命令行添加了一个-pa 'deps/bitcask/ebin'标识，这样当程序启动时，bertie就能自动载入bitcask的代码。

> **注意**　可以在git://github.com/joearms/bertie.git里下载整个示例。如果已经安装了rebar，你要做的就是下载这个项目并输入make。

25.3　生成依赖项本地副本

我的bertie应用程序在它的本地子目录里生成了bitcask的本地副本。有时候几个不同的应用程序会用到相同的依赖项，在这种情况下，我们会在应用程序的外部创建一个依赖项目录结构。

对于我的本地项目，我会把所有已下载的rebar依赖项保存在一个地方。我把所有这些依赖项保存在一个名为~joe/nobackup/erl_imports的顶级目录里。我这台机器的组织方式是nobackup目录下的任何文件都不会有备份。因为这些我感兴趣的文件在Web上到处都是，所以没有必要创建本地备份。

文件~joe/nobackup/erlang_imports/rebar.config列出了我想要使用的所有依赖项，它的内容如下：

```
{deps, [
  {cowboy,  ".*", {git, "git://github.com/extend/cowboy.git", "master"}},
  {ranch,   ".*", {git, "git://github.com/extend/ranch.git", "master"}},
  {bitcask, ".*", {git, "git://github.com/basho/bitcask.git", "master"}}
  ]}.
```

要获取这些依赖项，可以在保存配置文件的目录里输入命令`rebar get-deps`。

```
$ rebar get-deps
==> deps (get-deps)
Pulling cowboy from {git,"git://github.com/extend/cowboy.git","master"}
Initialized empty Git repository in /Users/joe/nobackup/deps/deps/cowboy/.git/
Pulling bitcask from {git,"git://github.com/basho/bitcask.git","master"}
Initialized empty Git repository in /Users/joe/nobackup/deps/deps/bitcask/.git/
==> cowboy (get-deps)
Pulling proper from {git,"git://github.com/manopapad/proper.git",{tag,"v1.0"}}
Initialized empty Git repository in /Users/joe/nobackup/deps/deps/proper/.git/
==> proper (get-deps)
==> bitcask (get-deps)
Pulling meck from {git,"git://github.com/eproxus/meck"}
Initialized empty Git repository in /Users/joe/nobackup/deps/deps/meck/.git/
==> meck (get-deps)
```

rebar不仅获取了在配置文件里指定的程序，还递归获取了这些程序所依赖的其他程序。

获取程序之后，我们用`rebar compile`命令来编译它们。

```
$ rebar compile
... 多行输出信息 ...
```

最后一步是把这些依赖项的保存位置告诉Erlang，具体做法是把下面这些代码行移至启动文件`${HOME}/.erlang`里：

```
%% 设置路径来让所有依赖项就位
Home = os:getenv("HOME").
Dir = Home ++ "/nobackup/erlang_imports/deps",
{ok, L} = file:list_dir(Dir).
lists:foreach(fun(I) ->
                  Path = Dir ++ "/" ++ I ++ "/ebin",
                  code:add_path(Path)
        end, L).
```

25.4　用 cowboy 构建嵌入式 Web 服务器

cowboy是一个小型、快速和模块化的HTTP服务器，它是用Erlang编写的，可以在https://github.com/extend/cowboy里找到。它得到了Nine Nines[①]公司的支持。

cowboy适合构建嵌入式应用程序。它没有配置文件，也不会生成日志。一切都是由Erlang控制的。

制作一个非常简单的Web服务器，它由命令`simple_web_server:start(Port)`启动。启动后的服务器会监听`Port`上的命令，它的根目录则是程序启动时的目录。

负责启动一切的主函数如下：

① http://ninenines.eu

cowboy/simple_web_server.erl

```
Line 1 start(Port) ->
     -     ok = application:start(crypto),
     -     ok = application:start(ranch),
     -     ok = application:start(cowboy),
     5     N_acceptors = 10,
     -     Dispatch = cowboy_router:compile(
     -                   [
     -                    %% {URIHost, list({URIPath, Handler, Opts})}
     -                    {'_', [{'_', simple_web_server, []}]}
    10                   ]),
     -     cowboy:start_http(my_simple_web_server,
     -                       N_acceptors,
     -                       [{port, Port}],
     -                       [{env, [{dispatch, Dispatch}]}]
    15                      ).
```

第2~4行启动OTP应用程序。第5行把Web服务器的“接收器”数量设为10，意思是Web服务器用10个并行进程来接受HTTP连接请求。同时进行的并行会话数可能会远超这个数字。Dispatch变量包含一个“调度器模式”列表，调度器模式会把URI路径映射到将要处理这个请求的模块名上。第9行的模式把所有请求都映射到了simple_web_server模块上。

cowboy_router:compile(Display)通过编译调度器信息来创建更高效的调度器，cowboy:start_http/4则会启动Web服务器。

调度器模式给出的模块必须提供三个回调方法：init/3、handle/3和terminate/2。这个案例里只有一个名为simple_web_server的处理模块。首先来看init/3，它会在Web服务器收到新连接时被调用。

cowboy/simple_web_server.erl

```
init({tcp, http}, Req, _Opts) ->
    {ok, Req, undefined}.
```

对init的调用包含三个参数。第一个说明了服务器收到的连接类型，在这个案例里是HTTP连接。第二个参数在cowboy里被称为请求对象，它包含了关于请求的信息，并将最终包含要发回浏览器的信息。cowboy提供了众多函数来从请求对象里提取信息，以及在请求对象里保存将被发回浏览器的信息。init的第三个参数（Opt）是调用cowboy_router:compile/1时在调度元组里给出的第三个参数。

init/3按惯例会返回元组{ok, Req, State}，使Web服务器接受这个连接。Req是请求对象，State是一个与连接相关的私有状态。如果连接被接受，HTTP驱动就会调用handle/2函数并附上init函数返回的请求对象和状态。handle/2的代码如下：

cowboy/simple_web_server.erl

```
handle(Req, State) ->
    {Path, Req1} = cowboy_req:path(Req),
```

```
3      Response = read_file(Path),
4      {ok, Req2} = cowboy_req:reply(200, [], Response, Req1),
5      {ok, Req2, State}.
```

handle调用cowboy_req:path(Req)（第2行）来提取被请求资源的路径。举个例子，如果用户请求来自http://localhost:1234/this_page.html这个地址的网页，cowboy_req:path(Req)就会返回路径<<"/this_page.html">>。路径由一个Erlang二进制型表示。

文件读取的结果（Response）通过cowboy_req:reply/4打包到请求对象里，成为handle/2返回值的一部分（第4~5行）。

读取被请求网页是由read_file/1完成的。

cowboy/simple_web_server.erl

```
read_file(Path) ->
    File = ["."|binary_to_list(Path)],
    case file:read_file(File) of
        {ok, Bin} -> Bin;
        _ -> ["<pre>cannot read:", File, "</pre>"]
    end.
```

因为假定所有输出的文件都来自Web服务器的启动目录，所以为文件名加了一个点作为前缀（否则会以一个斜杠开头），这样就能正确读取文件了。

基本上就是这样，下面发生的事将由套接字的建立方式决定。如果是一个长连接（keep-alive connection），handle就会被再次调用；如果连接被关闭，就会调用terminate/3。

cowboy/simple_web_server.erl

```
terminate(_Reason, _Req, _State) ->
    ok.
```

了解如何制作简单的服务器之后，我们将运用有关的基本结构来制作一个更有用的示例。

编写一个JSON往返程序，让数据从浏览器到Erlang再返回。这个示例的意义在于它能展示如何建立浏览器和Erlang的接口。从浏览器里的一个JavaScript对象入手，把它编码为JSON消息并发给Erlang，然后在Erlang里解码这个消息，将它变成Erlang数据结构，最后再发回浏览器并将它还原成一个JavaScript对象。如果一切顺利，这个对象就能顺利往返并保持原状。

先从Erlang代码开始，让它比之前例子里的更通用一些。添加一个元调用功能，这样就能从浏览器里调用任意Erlang函数了。当浏览器请求URI形式为http://Host/cgi?mod=Modname&Func=Funcname的网页时，我们想让Erlang Web浏览器调用函数Mod:Func(Args)，其中Args是JSON数据结构。

实现这种做法的代码如下：

cowboy/cgi_web_server.erl

```
handle(Req, State) ->
    {Path, Req1} = cowboy_req:path(Req),
    handle1(Path, Req1, State).
```

```
handle1(<<"/cgi">>, Req, State) ->
    {Args, Req1} = cowboy_req:qs_vals(Req),
    {ok, Bin, Req2} = cowboy_req:body(Req1),
    Val = mochijson2:decode(Bin),
    Response = call(Args, Val),
    Json = mochijson2:encode(Response),
    {ok, Req3} = cowboy_req:reply(200, [], Json, Req2),
    {ok, Req3, State};
handle1(Path, Req, State) ->
    Response = read_file(Path),
    {ok, Req1} = cowboy_req:reply(200, [], Response, Req),
    {ok, Req1, State}.
```

它与本章前面展示的代码很相似，只有少许区别：调用`cowboy_req:qs`来分解查询字符串，调用`cowboy_req:body`来提取HTTP请求的主体。还调用了来自`mochiweb2`库（位于https://github.com/mochi/mochiweb/）的编码与解码方法来实现JSON字符串和Erlang数据类型的相互转换。处理调用的代码如下：

cowboy/cgi_web_server.erl

```
call([{<<"mod">>,MB},{<<"func">>,FB}], X) ->
    Mod = list_to_atom(binary_to_list(MB)),
    Func = list_to_atom(binary_to_list(FB)),
    apply(Mod, Func, [X]).
```

这里是`echo`（回声）代码：

cowboy/echo.erl

```
-module(echo).
-export([me/1]).

me(X) ->
    io:format("echo:~p~n",[X]),
    X.
```

我们已经写完了Erlang代码，现在轮到浏览器里的对应代码了。它只需要几行JavaScript和jQuery库调用。

cowboy/test2.html

```
<script src="//ajax.googleapis.com/ajax/libs/jquery/1.9.1/jquery.min.js"></script>
<h1>Test2</h1>
<button id="button1">click</button>
<div id="result"></div>
<script>
$(document).ready(go);
var data = {int:1234,
            string:"abcd",
            array:[1,2,3,'abc'],
            map:{one:'abc', two:1, three:"abc"}};
```

```
function go(){
    $("#button1").click(test);
}
function test(){
    $.ajax({url:"cgi?mod=echo&func=me",
            type:'POST',
            data:JSON.stringify(data),
            success:function(str){
                var ret = JSON.parse(str);
                $("#result").html("<pre>"+
                                  JSON.stringify(ret, undefined, 4) +
                                  "</pre>");
            }});
}
</script>
```

现在在1234端口上启动Web服务器，这个操作可以在shell里进行。

```
1> cgi_web_server:start(1234).
{ok,...}
```

服务器启动之后，可以在浏览器里输入地址http://localhost:1234/test2.html，这样就能看见带有一个按钮的网页。点击按钮时会执行测试，浏览器会显示出从Erlang发回的数据（图25-1）。



图25-1 Erlang发回的JSON数据类型

在Erlang shell的窗口里能看到下面这些输出：

```
> echo:{struct,[{<<"int">>,1234},
                {<<"string">>,<<"abcd">>},
                {<<"array">>,[1,2,3,<<"abc">>]},
                {<<"map">>,
                {struct,[{<<"one">>,<<"abc">>},
                         {<<"two">>,1},
                         {<<"three">>,<<"abc">>}]}}]}
```

这是mochijson2:decode/1返回的Erlang解析树。如你所见，数据能在两个系统之间正确传输。

注意　不是所有JSON数据类型都能顺利往返于浏览器和Erlang之间。JavaScript的整数精度有限，而Erlang有大数字（bignum），所以在处理大整数时可能会遇到麻烦。类似地，浮点数可能在转换过程中损失精度。

除了从Erlang shell里启动Web服务器，我们还可能会想从makefile或命令行里启动服务器。在这种情况下，需要添加一个方法来把Erlang从shell里接收到的参数（一个原子列表）转换成启动服务器所需的格式。

cowboy/cgi_web_server.erl

```
start_from_shell([PortAsAtom]) ->
    PortAsInt = list_to_integer(atom_to_list(PortAsAtom)),
    start(PortAsInt).
```

举个例子，要启动一个监听5000端口的服务器，我们会给出以下命令：

```
$ erl -s cgi_web_server start_from_shell 5000
```

这一章展示了如何用rebar实现简单的项目管理。展示了如何在GitHub创建一个新项目，如何用rebar管理它，如何包含来自GitHub的项目，以及如何把它们包含在自己的项目里。我们用一个简单的例子展示了如何用cowboy构建一个专门的Web服务器。在第18章里构建的Web服务器和本章展示的代码非常相似。

第二个cowboy示例用来自mochiweb的编码和解码方法把JSON数据类型转换成Erlang结构。当Erlang的R17版引入映射组后，我将修改本书里的代码来反映这一变化。

在下一章里，我们将来了解多核计算机，同时探索一些并行化技巧来让代码运行在多核计算机上。当在多核CPU上运行时，并发程序会变成并行程序，所以运行速度应当会更快——看看是否真是这样。

25.5　练习

(1) 注册一个GitHub账号，然后按照本章开头的步骤来创建你自己的项目。

(2) 第二个cowboy示例可能是不安全的。用户可以通过CGI调用接口请求执行任意的Erlang

模块。重新设计这个接口，让它只允许调用一组事先定义的模块。

(3) 对cowboy示例做一次安全审计，因为它的代码里有许多安全问题。例如被请求文件的值未经检查，这样用户就能访问Web服务器目录结构以外的文件。找到并修复这些安全问题。

(4) 任何主机都能连接到cowboy服务器。修改它的代码，让它只允许来自已知IP地址的主机连接。把这些主机保存到某种持久性数据库里，比如Mnesia或bitcask。记录某个特定主机进行了多少次连接。制作一个主机黑名单，登记那些在给定时间段内连接过于频繁的主机。

(5) 修改Web服务器，让它允许对通过CGI接口调用的模块进行动态重编译。在我们的示例里，`echo.erl`模块必须先编译才能调用。当某个模块通过CGI接口被调用时，读取其beam文件的时间戳并与对应`.erl`的时间戳进行比较，如有必要就重新编译和载入Erlang代码。

(6) rebar是把Erlang程序作为“独立”二进制文件分发的优秀范例。请把rebar的可执行文件复制到一个空白目录并重命名为`rebar.zip`（rebar其实是一个zip文件），然后解压缩并检查里面的内容。用cowboy示例代码制作你自己的自执行二进制文件。

第26章 多核CPU编程

如何编写能在多核CPU上跑得更快的程序？答案尽在可变状态和并发里。

在过去（大约20年之前），并发有两种模式：

- 共享状态式并发；
- 消息传递式并发。

编程界选择了一个方向（通往共享状态），Erlang社区则选择了另一个方向。（还有零星几种编程语言选择了“消息传递式并发”这条路，比如Oz和Occam。）

消息传递式并发里不存在共享状态，所有计算都是在各个进程里完成的，异步消息传递是唯一的数据交换方式。

为什么这是件好事？

共享状态式并发涉及“可变状态”（可修改的内存）这个概念，所有语言（比如C、Java和C++）都有一种叫状态的东西，而且我们可以改变它。

如果只有一个进程能改变它，倒也没什么问题。

但如果你有多个进程共享和修改同一处内存区域，就有大麻烦了，会发生许多疯狂的事。

为了防止同时修改共享内存，我们会使用一种锁定机制。你可以叫它互斥（mutex）、同步方法（synchronized method）或其他什么名字，但它归根到底就是一种锁。

如果程序在临界区崩溃了（当它们持有锁时），灾难就会发生，其他所有程序都不知道该怎么办。如果程序损坏了共享状态的内存，也会发生灾难，其他程序同样不知道该怎么办。

程序员该如何修复这些问题？恐怕会大费周折。他们的程序也许在单核处理器上能用，但对多核处理器来说，这简直是灾难。

这类问题有多种解决方案（事务内存很可能是首选），但它们在最好的情况下也不过是勉强解决问题，在最坏的情况下则会是梦魇。

Erlang没有可变的数据结构（这句话不是百分百正确，但算是基本正确）。

- 没有可变的数据结构 = 没有锁。
- 没有可变的数据结构 = 能够轻松并行。

如何实现并行计算？很简单。程序员把问题的解决方案分配到若干个并行进程里。

这种编程风格有它自己的专业术语：面向并发编程。

26.1　给 Erlang 程序员的好消息

这里有个好消息：你的Erlang程序也许能在n核处理器里快上n倍，而且无需修改程序。

但是必须遵循一套简单的规则。

如果想让应用程序在多核CPU里运行得更快，就必须确保它拥有大量互不冲突的进程，而且程序里没有顺序瓶颈。

如果编写的是一整块顺序代码，并且没有使用`spawn`来创建哪怕一个并行进程，程序也许就不会变得更快。

不要丧气。即使你的程序一开始是一个巨大的顺序程序，对它做一些简单的修改也能使它并行化。

在本章，我们将看到以下主题：

- 如何让程序高效运行在多核CPU上；
- 如何让顺序程序并行化；
- 顺序瓶颈的问题；
- 如何避免副作用。

介绍完这些之后，我们将来看一个更复杂的设计问题。我们将实现一个名为`mapreduce`的高阶函数，并展示如何用它来使计算并行化。`mapreduce`是由谷歌公司开发的一种抽象，用来在多组计算单元上执行并行计算。

26.2　如何在多核 CPU 上使程序高效运行

要实现高效运行，必须做下面这些事：

- 使用大量进程；
- 避免副作用；
- 避免顺序瓶颈；
- 编写“小消息，大计算”的代码。

如果都做到了，我们的Erlang程序就应当能高效运行在多核CPU上。

为什么要重视多核CPU

你可能会疑惑这么大费周章是为什么。难道非得为了多核运行而让程序并行化吗？答案是肯定的。如今，带有超线程的四核CPU十分常见，许多智能手机都是四核的。甚至我的低配置MacBook Air都有超线程的双核CPU，而我的台式机是超线程的八核。

让程序在双核机器里快上两倍不算太惊人（但也有一点点惊人）。不过还是别自欺欺人了，双核处理器的时钟速度比单核CPU要慢一些，所以性能的提升也许很有限。

两倍不会让我感到兴奋，十倍却会，一百倍则会让我非常、非常兴奋。现代处理器的速度是如此之快，一个核心就能运行4个超线程，因此一个32核的CPU也许就能带给我们128个线程。这就意味着快上一百倍的速度触手可及。

> 快上一百倍真的会让我很兴奋。
> 我们要做的就是编写代码。

26.2.1　使用大量进程

这一点很重要，我们必须让CPU保持忙碌。所有的CPU在任何时候都应当是忙碌的。要实现这一点，最简单的方式就是使用大量进程。

当我说大量时，我的意思是相对于CPU数量的大量。如果有大量的进程，就不需要担心CPU能否保持忙碌了。这看起上去纯粹是一种统计效应。如果只有少量进程，那么有时候它们可能只占用了其中一个CPU。如果有大量的进程，这种效应似乎就会消失。如果想让程序为未来做好准备，就必须考虑到虽然今天的芯片里只有少量CPU，但未来的芯片里可能会有几千个CPU。

这些进程如果有着相似的工作量就更好了。把程序编写成某个进程忙碌而其余进程空闲可不是个好主意。

我们会在许多应用程序里“免费”得到大量进程。如果应用程序是“天生并行”的，就无需操心让代码并行化了。举个例子，如果正在编写一个同时管理几万个连接的消息传输系统，就能获得这几万个连接的并发，而处理单个连接的代码无需操心并发性。

26.2.2　避免副作用

副作用会妨碍并发性。我们在本书的开头部分就讨论了“不会变的变量”，这是理解Erlang程序为什么能在多核CPU上跑得比其他程序（用能破坏性修改内存的语言编写）更快的关键。

在带有共享内存和线程的语言里，如果两个线程同时写入公共内存，灾难就可能发生。实现共享内存式并发的系统通过给正在写入的共享内存加锁来避免这种事情发生。这些锁对程序员隐藏，在不同的语言里表现为互斥或同步方法。共享内存的主要问题在于一个线程可以损坏另一个线程所使用的内存。因此，即使我的程序是正确的，另一个线程也可能会弄乱我的数据结构，导致程序崩溃。

Erlang没有共享内存，所以不存在这个问题。事实上，这句话并不完全正确。只有两种方式能共享内存，所以这个问题可以轻松避免。这两种共享内存的方式和共享式ETS或DETS表有关。

不要使用共享式ETS或DETS表

ETS表可以被多个进程共享。我们在19.3节里介绍了创建ETS表的几种方法。通过在ets:new里使用某个选项，就能创建出一个public类型的表。如果你还记得的话，它的效果如下：

> 创建一个公共表，任何知道此表标识符的进程都能读取和写入这个表。

这么做可能很危险，只有在以下条件得到满足时才是安全的：

- 每次只能有一个进程写入表，其他进程可以读取表；
- 写入ETS表的进程是正确的，不会把错误数据写入表。

系统一般不能满足这些条件，而是要靠程序逻辑。

- 注意1：ETS表的每一种操作都是原子式的，但一系列ETS操作无法作为一个原子单元执行。虽然不会损坏ETS表里的数据，但如果多个进程试图同时更新一个共享表而又没有协调好，就可能出现逻辑上不一致的表。
- 注意2：ETS表类型protected的安全性要高得多。只有一个进程（即所有者）能写入表，但可以有多个进程读取这个表。这一点由系统保证。但是请记住，即使只有一个进程能写入ETS表，如果它损坏了表里的数据，也会影响到所有读取这个表的进程。

如果你使用的ETS表类型是private，那么你的程序就是安全的。上述结论也适用于DETS。可以创建能被多个不同进程写入的共享式DETS表，但不鼓励这种做法。

注意 ETS和DETS原本并不是为了独立使用而创建的，而是为了实现Mnesia。原本的意图是如果应用程序想要模拟进程间共享内存，就应该使用Mnesia的事务机制。

26.2.3 避免顺序瓶颈

当我们使程序并行化，确保有大量进程并且没有共享内存操作后，接下来的问题就是思考顺序瓶颈。有些事情本来就是顺序的，如果问题本身具有“顺序性”，我们是无法改变这一点的。某些事件会按照顺序发生，无论我们有多努力，都无法改变这个顺序。我们出生、活着，然后死亡。我们改变不了这个顺序，无法让这些事情并行化。

顺序瓶颈指的是多个并发进程需要访问某种顺序资源。一个典型的例子是I/O。通常只有一个磁盘，对这个磁盘的所有输出最终都将是顺序的。这个磁盘只有一组磁头，而不是两组，我们无法改变这一点。

每次创建注册进程时都可能形成一个顺序瓶颈，所以要尽量避免使用注册进程。如果需要创建一个注册进程并把它当作服务器，就要确保它尽可能快地响应所有请求。

在很多时候，解决顺序瓶颈的唯一方式就是改变相关的算法，没有其他捷径可走。必须把非分布式算法修改成分布式算法。这个主题（分布式算法）有大量的研究资料可供参考，但是在传统的编程语言库里应用较少。低使用率的主要原因是以往对这类算法的需求不明显，直到为网络或多核计算机编写算法时才有所改观。

为多核CPU或永久连接互联网的计算机编程将迫使我们深入研究这些资料，并实现其中一些很不错的算法。

一种分布式订票系统

假设我们有一种资源：一批下一场Strolling Bones演唱会的票。为了确保买票后能真正得到票，传统上会使用单个代理来订购所有的票，但这会引入一个顺序瓶颈。

要避免单个代理的瓶颈，可以设置两个订票代理。在销售开始时，第一个订票代理会得到所有偶数编号的票，第二个订票代理会得到所有奇数编号的票。通过这种方式，就能确保代理们不会两次销售同一张票。

如果其中一个代理卖完了票，它可以向另一个代理请求一批票。

我并不是说这是一种好方法，因为你去看演唱会时可能会想坐在朋友旁边。但是把售票处一分为二的做法确实消除了瓶颈。

这种把单个订票代理替换成*n*个分布式代理（*n*可以随时间而变），并且各个代理可以加入和离开售票网络以及随时崩溃的做法是当前热门的分布式计算研究领域。这个研究领域被称为分布式散列表（distributed hash tables）。如果搜索这个名词，就能找到大量的相关资料。

26.3　让顺序代码并行

还记得我们着重介绍过的一次一列表操作（特别是`lists:map/2`函数）吗？`map/2`的定义如下：

```
map(_, [])    -> [];
map(F, [H|T]) -> [F(H)|map(F, T)].
```

一种让顺序程序加速的简单策略是用新版的`map`（我称之为`pmap`）来取代所有`map`调用，它会并行执行所有的参数。

lib_misc.erl

```
pmap(F, L) ->
    S = self(),
    %% make_ref() 返回一个唯一的引用，
    %% 稍后会匹配它
    Ref = erlang:make_ref(),
    Pids = map(fun(I) ->
                       spawn(fun() -> do_f(S, Ref, F, I) end)
               end, L),
    %% 收集结果
    gather(Pids, Ref).

do_f(Parent, Ref, F, I) ->
    Parent ! {self(), Ref, (catch F(I))}.
gather([Pid|T], Ref) ->
    receive
        {Pid, Ref, Ret} -> [Ret|gather(T, Ref)]
    end;
gather([], _) ->
    [].
```

`pmap`的工作方式类似于`map`，但是当调用`pmap(F, L)`时，它会为`L`里的每个参数分别创建一个并行进程然后执行。请注意，执行`L`里各个参数的进程没有固定的完成顺序。

`gather`函数里的选择性`receive`会确保返回值里的参数顺序符合原始列表的顺序。

`map`和`pmap`有一点语义上的区别。在`pmap`里，我们使用`catch F(H)`来把函数映射到列表上，而在`map`里直接用`F(H)`。这是因为想要确保`pmap`能正确终止，避免发生计算`F(H)`时抛出异常错误这种情况。如果没有抛出异常错误，这两个函数的行为就是相同的。

提示 最后这句话严格来讲并非完全正确。如果map和pmap有副作用，那么它们的行为就会有所不同。假设F(H)包含一些修改进程字典的代码。调用map时，对进程字典的修改会发生在调用map的进程里。

而当我们调用pmap时，每个F(H)都在它自己的进程里执行，所以如果使用了进程字典，对字典的修改就不会影响到调用pmap的进程。

警告 具有副作用的代码不能简单地用pmap取代map调用来实现并行。

何时使用pmap

用pmap代替map并不是加速程序的万能药。下面是一些需要考虑的事项。

1. 并发粒度

如果函数要做的工作很少就别用pmap。假设我们要做这个：

```
map(fun(I) -> 2*I end, L)
```

这个fun的工作量很小。建立进程并等待回复的开销会超过用并行进程完成任务所带来的益处。

2. 不要创建太多进程

别忘了pmap(F, L)会创建length(L)个并行进程。如果L非常大，就会创建大量进程。最好的做法是创建lagom数量的进程。Erlang来自瑞典，lagom这个词可以大致翻译成“不太少，也不太多，刚刚好”。有人认为这总结了瑞典人的性格。

3. 思考你需要的抽象

pmap可能不是合适的抽象。有许多方式可用来把函数并行映射到列表上，这里选择的是最简单的一种。

刚才使用的pmap版本关心返回值里的元素顺序（用选择性接收来处理）。如果不关心返回值的顺序，就可以这么写：

lib_misc.erl

```
pmap1(F, L) ->
    S = self(),
    Ref = erlang:make_ref(),
    foreach(fun(I) ->
                    spawn(fun() -> do_f1(S, Ref, F, I) end)
            end, L),
    %% 收集结果
    gather1(length(L), Ref, []).
```

```
do_f1(Parent, Ref, F, I) ->
    Parent ! {Ref, (catch F(I))}.

gather1(0, _, L) -> L;
gather1(N, Ref, L) ->
    receive
        {Ref, Ret} -> gather1(N-1, Ref, [Ret|L])
    end.
```

做一些简单的修改就能把它变成一个并行的foreach。它的代码类似于前面这个，但我们不会构建任何返回值，只会标注程序结束。

另一种方式是用最多K个进程来实现pmap，其中K是一个固定常数。如果想把pmap用于非常大的列表，那么这种方式可能会很有用。

另一个版本的pmap可以把计算映射到分布式网络的节点上，而不仅仅是多核CPU的各个进程上。

我不会在这里展示如何实现它们，你可以自己思考一下。

本节的目的是指出你可以用基本函数spawn、send和receive来轻松构建众多抽象。你可以用这些基本函数创建自己的并行控制抽象，从而提升程序的并发性。

和前面一样，避免副作用是提升并发性的关键。永远别忘了这一点。

26.4　小消息，大计算

谈过理论之后，现在来实际测量一下。在这一节里，我们将进行两个实验。把两个函数分别映射到一个包含100个元素的列表上，然后比较并行映射和顺序映射各自花费的时间。

我们将使用两组不同的问题。第一组会计算：

```
L = [L1, L2, ..., L100],
map(fun lists:sort/1, L)
```

L里的每个元素都是一个包含1000个随机整数的列表。

第二组会计算：

```
L = [27,27,..., 27],
map(fun ptests:fib/1, L)
```

这里的L是一个包含100个27的列表，我们将计算列表[fib(27), fib(27), ...]（fib是斐波纳契函数）。

给这两个函数计时，然后用pmap替换map并再次计时。

在第一个（排序）计算里使用pmap涉及在不同进程间发送比较大的数据（包含1000个随机整数的列表），但排序过程是相当快的。第二个计算涉及给各个进程发送一个很小的请求（来计算fib(27)），但递归计算fib(27)的运算量相对较大。

因为计算fib(27)涉及的进程间数据复制很少，而运算量相对较大，所以预计第二组问题在多核CPU上的性能会胜过第一组。

要了解它们的实际效果，需要一个脚本来进行自动化测试。不过，首先来看如何启动SMP Erlang。

运行SMP Erlang

对称多处理（Symmetric Multiprocessing，简称SMP）的机器有两个或更多个相同的CPU连接到同一块共享内存。这些CPU可以属于同一块多核芯片，也可以分属多块芯片，或者两者皆有。Erlang能在许多不同的SMP架构和操作系统上运行。当前系统运行在支持一个或两个处理器的主板上，用的是Intel双核和四核处理器。它还能用Sun和Cavium处理器运行。这个领域的开发速度极快，每个版本的Erlang都增加了一些操作系统和处理器支持。你可以在当前Erlang分发套装的发布说明里找到最新信息。（点击http://www.erlang.org/download.html下载目录里的最新版Erlang标题。）

注意 从R11B-0版的Erlang开始，SMP Erlang是默认启用的（也就是说默认会构建SMP虚拟机）。要在其他平台上强制构建SMP Erlang，就应当给`configure`命令加上`--enable-smp-support`标记。

SMP Erlang有两个命令行标记，它们将决定如何在多核CPU上运行。

```
$erl -smp +S N
```

- *-smp*
  启动SMP Erlang。
- *+S N*
  用N个调度器运行Erlang。每个Erlang调度器都是一个完整的虚拟机，并且完全了解其他所有虚拟机。如果省略了这个参数，就会默认设为SMP机器上的逻辑处理器数量。

为何会想要修改这个值？原因如下。

- 测量性能时，有时会想要改变调度器的数量来看看不同数量CPU的运行效果。
- 通过修改N，可以在单核CPU上模拟多核CPU的运行。
- 我们也许会希望调度器的数量超过物理处理器的数量。这么做有时候能提升吞吐量，让系统表现得更好。这些效果尚未得到充分理解，属于活跃的研究领域。

要执行测试，需要一个脚本来运行它们。

runtests

```
#!/bin/sh
echo "" >results
for i in 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16\
        17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32
```

```
do
   echo $i
   erl -boot start_clean -noshell -smp +S $i \
       -s ptests tests $i >> results

done
```

这段代码会用32个不同的调度器启动Erlang，运行计时测试，然后把计时结果归集到一个名为`results`的文件里。

然后需要一个测试程序。

ptests.erl

```
-module(ptests).
-export([tests/1, fib/1]).
-import(lists, [map/2]).
-import(lib_misc, [pmap/2]).

tests([N]) ->
    Nsched = list_to_integer(atom_to_list(N)),
    run_tests(1, Nsched).

run_tests(N, Nsched) ->
    case test(N) of
        stop ->
            init:stop();
        Val ->
            io:format("~p.~n",[{Nsched, Val}]),
            run_tests(N+1, Nsched)
    end.
test(1) ->
    %% 生成100个列表，
    %% 每个列表都包含1000个随机整数
    seed(),
    S = lists:seq(1,100),
    L = map(fun(_) -> mkList(1000) end, S),
    {Time1, S1} = timer:tc(lists,    map,  [fun lists:sort/1, L]),
    {Time2, S2} = timer:tc(lib_misc, pmap, [fun lists:sort/1, L]),
    {sort, Time1, Time2, equal(S1, S2)};
test(2) ->
    %% L = [27,27,27,..]，共100个
    L = lists:duplicate(100, 27),
    {Time1, S1} = timer:tc(lists,    map,  [fun ptests:fib/1, L]),
    {Time2, S2} = timer:tc(lib_misc, pmap, [fun ptests:fib/1, L]),
    {fib, Time1, Time2, equal(S1, S2)};
test(3) ->
    stop.

%% Equal用于测试map和pmap的计算结果是否相同
```

```
equal(S,S)   -> true;
equal(S1,S2) ->  {differ, S1, S2}.

%% 递归（低效率）的斐波那契
fib(0) -> 1;
fib(1) -> 1;
fib(N) -> fib(N-1) + fib(N-2).

%% 重置随机数生成器，
%% 这样每次运行程序都能
%% 获得相同的随机数列

seed() -> random:seed(44,55,66).

%% 生成包含K个随机数的列表，
%% 每个随机数都在1..1000000之间
mkList(K) -> mkList(K, []).

mkList(0, L) -> L;
mkList(N, L) -> mkList(N-1, [random:uniform(1000000)|L]).
```

这段代码会在两个测试案例里分别运行`map`和`pmap`。结果如图26-1所示，我们用点标绘了`pmap`和`map`所用时间之比。

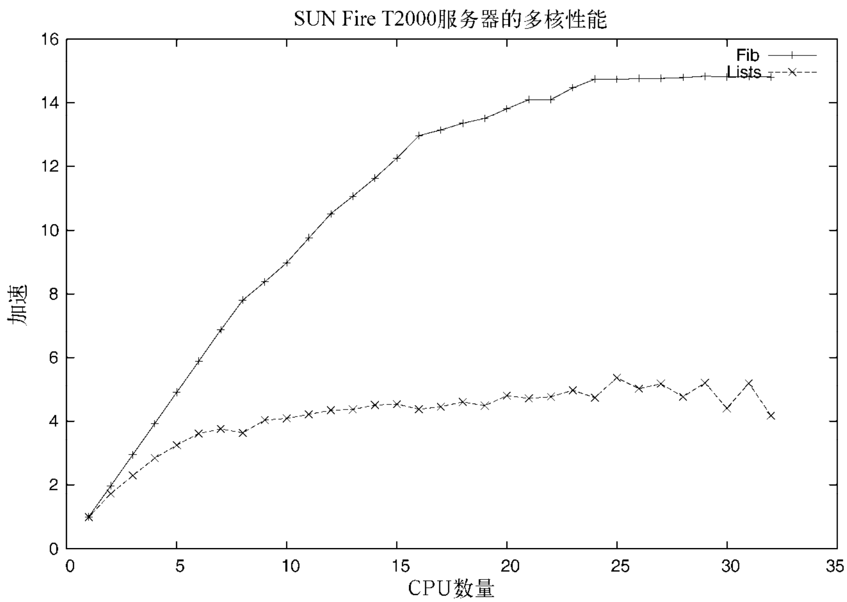


图 26-1

如你所见，消息传递少的CPU密集型计算能够线性加速，而消息传递多的轻量型计算就没有那么好的可伸缩性了。

最后需要说明的是，不应该过度解读这些图表。SMP Erlang每天都在改进，所以今天是事实不代表明天还是事实。但是，可以说这些结果非常鼓舞人心。爱立信正在构建的商业产品能在双核处理器上跑出近乎两倍的速度，所以我们非常开心。

26.5　用 mapreduce 使计算并行化

现在我们将把理论转化成实践。首先来看高阶函数mapreduce，然后展示如何用它来让一种简单的计算并行化。

mapreduce

通过图26-2来了解一下mapreduce的基本概念。图中这些映射（map）进程会生成由{Key, Value}对组成的消息流，并发送给一个化简（reduce）进程进行合并，后者会把具有相同键的对组合在一起。

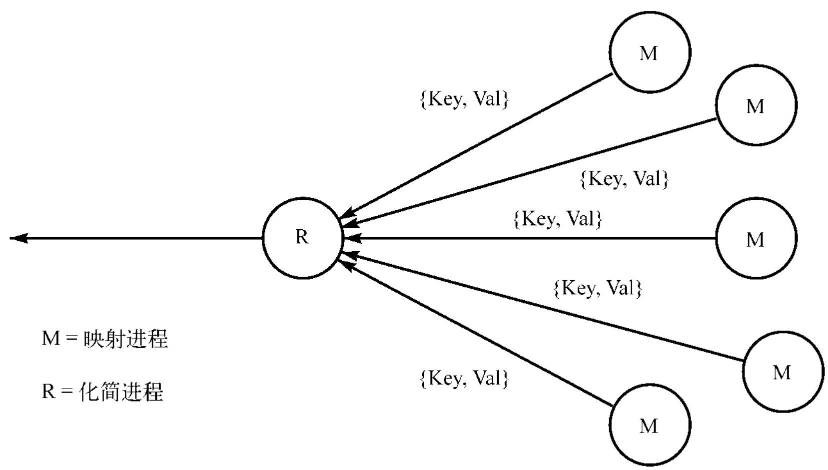


图　26-2

警告　mapreduce上下文里的map和本书其他地方出现的map函数完全不同。

mapreduce是一个并行高阶函数。它是由谷歌公司的Jeffrey Dean和Sanjay Ghemawat提出的，据说谷歌的服务器集群每天都在使用它。

可以用许多不同的方式和语义实现mapreduce，它其实更像是一系列算法，而不是某种具体算法。

mapreduce的定义如下：

```
-spec mapreduce(F1, F2, Acc0, L) -> Acc
    F1 = fun(Pid, X) -> void
    F2 = fun(Key, [Value], Acc0) -> Acc
    L  = [X]
    Acc = X = term()
```

- F1(Pid, X)是映射函数。

  F1的任务是发送一个{Key, Value}消息流到Pid进程，然后终止。mapreduce会为列表L里的每一个X值分裂出一个全新进程。
- F2(Key, [Value], Acc0) -> Acc是化简函数。

  当所有映射进程都终止时，化简函数就应该已经合并某个键的所有值了。mapreduce随后对收集的各个{Key, [Value]}元组调用F2(Key, [Value], Acc)。Acc是一个初始值为Acc0的归集器，F2会返回一个新的归集器。（另一种描述方式是F2对收集的{Key, [Value]}元组执行了折叠操作。）

  Acc0是归集器的初始值，在调用F2时使用。
- L是一个由X值组成的列表。

  F1(Pid, X)会被用于L里的每一个X值。Pid是化简进程的标识符，它是由mapreduce创建的。

mapreduce是在phofs（parallel higher-order functions的简写，即并行高阶函数）模块里定义的。

phofs.erl

```
-module(phofs).
-export([mapreduce/4]).
-import(lists, [foreach/2]).

%% F1(Pid, X) -> 发送 {Key,Val} 消息到 Pid
%% F2(Key, [Val], AccIn) -> AccOut
mapreduce(F1, F2, Acc0, L) ->
    S = self(),
    Pid = spawn(fun() -> reduce(S, F1, F2, Acc0, L) end),
    receive
        {Pid, Result} ->
            Result
    end.

reduce(Parent, F1, F2, Acc0, L) ->
    process_flag(trap_exit, true),
    ReducePid = self(),
    %% 创建一些Map进程，
    %% 分别用于L里的各个X元素
    foreach(fun(X) ->
                    spawn_link(fun() -> do_job(ReducePid, F1, X) end)
            end, L),
    N = length(L),
```

```
    %% 生成一个字典来保存各个Key
    Dict0 = dict:new(),
    %% 等待N个Map进程终止
    Dict1 = collect_replies(N, Dict0),
    Acc = dict:fold(F2, Acc0, Dict1),
    Parent ! {self(), Acc}.

%% collect_replies(N, Dict)
%% 收集与合并来自N个进程的{Key, Value}消息
%% N个进程都终止后返回一个包含
%% {Key, [Value]}元组的字典

collect_replies(0, Dict) ->
    Dict;
collect_replies(N, Dict) ->
    receive
        {Key, Val} ->
            case dict:is_key(Key, Dict) of
                true ->
                    Dict1 = dict:append(Key, Val, Dict),
                    collect_replies(N, Dict1);
                false ->
                    Dict1 = dict:store(Key,[Val], Dict),
                    collect_replies(N, Dict1)
            end;
        {'EXIT', _,  _Why} ->
            collect_replies(N-1, Dict)
    end.

%% Call F(Pid, X)
%% F必须发送{Key, Value}
%% 消息到Pid，然后终止

do_job(ReducePid, F, X) ->
    F(ReducePid, X).
```

进入下一步之前，我们将测试mapreduce来彻底弄明白它的工作方式。

编写一个小程序来统计本书所附代码目录里所有单词的出现次数。这个程序如下：

test_mapreduce.erl

```
-module(test_mapreduce).
-compile(export_all).
-import(lists, [reverse/1, sort/1]).

test() ->
    wc_dir(".").

wc_dir(Dir) ->
```

```
    F1 = fun generate_words/2,
    F2 = fun count_words/3,
    Files = lib_find:files(Dir, "*.erl", false),
    L1 = phofs:mapreduce(F1, F2, [], Files),
    reverse(sort(L1)).

generate_words(Pid, File) ->
    F = fun(Word) -> Pid ! {Word, 1} end,
    lib_misc:foreachWordInFile(File, F).
count_words(Key, Vals, A) ->
    [{length(Vals), Key}|A].
```

```
1> test_mapreduce:test().
[{341,"1"},
 {330,"end"},
 {318,"0"},
 {265,"N"},
 {235,"X"},
 {214,"T"},
 {213,"2"},
 {205,"start"},
 {196,"L"},
 {194,"is"},
 {185,"file"},
 {177,"Pid"},
 ...
```

当我运行它时，代码目录里有102个Erlang模块。mapreduce创建了102个并行进程，每个进程都向化简进程发送了包含对的消息流。它应该能在100核的处理器上流畅运行（如果磁盘能跟得上的话）。

刚开始研发Erlang的时候（1985年），根本没想到并行计算机会随处可见，也没想到计算机集群可以放入单个芯片。Erlang被设计用来编写容错式计算机集群。要实现容错，必须拥有一台以上的机器，而且程序必须兼顾并发和并行运行。

多核CPU出现以后，我们发现许多程序的速度直接变快了。这些程序是用并行执行的方式编写的，我们要做的就是用并行硬件来运行它们。

Erlang没有提供pmap和mapreduce这样的并行抽象，而是提供了一小组基本函数（spawn、send和receive）来帮助你构建这些抽象。这一章展示了如何用底层基本函数来构建pmap这样的事物。

到目前为止，我们已经在本书里介绍了许多基本知识，包括标准库和OTP框架的主要组成部分，如何制作一个独立的系统，以及如何在分布式系统和多核CPU上构建程序。

在下一章将直接进入一个应用程序，它会用到在本书前面开发的许多技术。我们还将解决福尔摩斯的最后一案。

26.6　练习

让计算并行化往往能大大加快响应速度，我们将在下面这些例子里展示这一点。

(1) 我们在17.1.1节里编写了一个获取网页的程序。修改这个程序，用`HEAD`命令取代`GET`命令。可以通过发送HTTP `HEAD`命令来测量网站的响应时间。服务器响应`HEAD`命令时只会返回网页的头部，不会返回主体。编写一个名为`web_profiler:ping(URL, Timeout)`的函数来测量URL这个网站地址的响应速度。它应当返回`{time, T}`或者`timeout`。

(2) 制作一个包含大量网站的列表`L`，记录`lists:map(fun(I) -> web_profiler:ping(URL, Timeout) end, L)`所花费的时间。它也许会运行很久，最坏情况下是`Timeout x length(L)`。

(3) 用`pmap`代替`map`重复刚才的计时。现在所有的`HEAD`请求都应当是并行的。因此，最坏情况下的响应时间就是`Timeout`。

(4) 把结果保存在一个数据库里，然后制作一个Web接口来查询这个数据库。可以从第24章里开发的数据库和Web服务器代码入手。

(5) 如果你用一个超长的元素列表调用`pmap`，就可能会创建出过多的并行进程。编写一个名为`pmap(F, L, Max)`的函数来并行计算列表`[F(I) || I <- L]`，但限制它同时最多只能运行`Max`个并行进程。

(6) 编写一个新版的`pmap`，让它能工作在分布式Erlang上，把任务分派给多个Erlang节点。

(7) 编写一个新版的`pmap`，让它能工作在分布式Erlang上，把任务分派给多个Erlang节点，并且实现各个节点之间的任务负载均衡。

第 27 章

福尔摩斯的最后一案

“我们有一小块程序碎片，”阿姆斯特朗说，“但是不知道它来自哪里或者写的是什么。”

“给我看看。”福尔摩斯说。

“就是这个。”阿姆斯特朗说。他俯下身子靠近福尔摩斯，向他展示一张纸片，上面写满各种奇怪的符号、括号和箭头，还掺杂着一些英文文本。

“这是什么？”阿姆斯特朗问，“雷斯垂德说它是一种来自未来的强力黑魔法。”

“这是计算机程序的一部分，”福尔摩斯一边说一边抽着烟斗，“我在时间旅行中曾经设法从未来传回过一台计算机和大量文件。根据这些文件，我发现Erlang的邮件列表里有73 445封邮件。我想，要是把纸片上的文字和这些邮件进行对比，就能找出这些奇怪符号的意思。但是该怎么做呢？我取出小提琴，弹奏了一段帕格尼尼的曲子，然后有了思路。最接近纸片文字的邮件列表项必然能让文档词汇的TF*IDF分数具有最大的余弦相似度……”

“很精彩，”阿姆斯特朗说，“但什么是TF*IDF分数？”

“这很简单，我亲爱的阿姆斯特朗，”福尔摩斯说，“它是词汇频度乘以逆文档频度。我来解释一下……”

27.1 找出数据的相似度

在写作本书时，Erlang的邮件列表[①]里包含73 445封邮件。它们代表了庞大的知识储备，可以用多种方式加以利用。我们可以用它来回答关于Erlang的问题，也可以从中寻找灵感。

如果你正在编写一个程序并且需要帮助，就可以找sherlock（福尔摩斯）帮忙。sherlock会分析你的程序，从Erlang列表里找出现存最相似的邮件，然后把这些可能有用的资料提供给你。

现在来看看总体方案。第一步是下载整个Erlang邮件列表然后本地存储它。第二步是组织并解析所有的邮件。第三步是计算邮件的某些属性，以便进行相似度搜索。第四步是进行邮件查询来找到与你的程序最相似的邮件。

这就是本章的核心概念。具体的做法有很多种，可以下载现存的所有Erlang模块然后进行相似度搜索，也可以下载一个庞大的推特集合，或者任何感兴趣的数据集。

① http://erlang.org/pipermail/erlang-questions/

还有一种方法可以帮助你归类数据。假设你刚刚写了一小段笔记，想把它保存在一个文件里。决定用什么文件名或目录来保存这个文件是个难题。也许它和几年前我在磁盘里写的文件很相似，或者某个我根本不认识的人也写过类似的文件。在这些情形下，我希望能从一个极其庞大的文档集合里找出最接近它的文档。

sherlock能在我们不知道存在相似性时找出事物之间的相似性。我们用它来寻找Erlang邮件列表里的相似内容。

深入sherlock的实现细节之前，先来展示一下它的威力。

27.2　sherlock 演示

在这一节里，我们将试用一下sherlock。首先必须初始化sherlock，做法是从Erlang的邮件列表存档里获取并分析数据。做完这些之后，就能以多种方式查询数据了。

27.2.1　获取并预处理数据

这一节里的某些命令会花费很长时间，所以只执行一次。首先来初始化系统。

```
1> sherlock:init().
Making ${HOME}/.sherlock/mails
sherlock_cache_created
```

这个命令会在你的主目录下创建一个目录结构。如果环境变量`HOME`未设置，它就会失败。sherlock会把所有数据都保存在顶级目录`${HOME}/.sherlock`下的目录结构里。

```
2> sherlock:fetch_index().
Written: /Users/joe/.sherlock/mails/questions.html
Written: /Users/joe/.sherlock/mails/questions.term
176 files must be fetched
```

`fetch-index`会获取Erlang邮件列表里的邮件索引。该索引是一个HTML文件，我们可以从中提取出所有保存邮件的文件名列表。

```
3> sherlock:fetch_mails().
fetching:"http://erlang.org/pipermail/erlang-questions/1997-January.txt.gz"
written:/Users/joe/.sherlock/mails/cache/1997-January.txt.gz
fetching:"http://erlang.org/pipermail/erlang-questions/1997-May.txt.gz"
written:/Users/joe/.sherlock/mails/cache/1997-May.txt.gz
...
```

有了这些文件名，我们要做的就是下载它们。每个文件名都包含年和月。把下载的所有文件都保存在名为`${HOME}/.sherlock/mail/cache`的目录里，这些数据应该保留到不再运行程序为止。我在写作这一章并运行前面的命令后一共下载了176个文件（37MB的压缩邮件数据），它们代表了73 445封邮件。

可以向sherlock了解已收集数据的信息。

```
4> sherlock:mail_years().
["1997","1998","1999","2000","2001","2002","2003","2004",
"2005","2006","2007","2008","2009","2010","2011","2012",
"2013"]
```

这说明已经成功下载了从1997年到2013年的邮件。

下一步是把数据按年归类，我们将为每个年份创建一个新目录。例如，2007年的数据会被保存在`${HOME}/.sherlock/mail/2007`目录里，以此类推。接下来会集中每一年的所有邮件，然后把结果写入合适的目录。做完这些之后，我们将解析并后处理每一年的数据。

```
5> sherlock:process_all_years().
Parsing mails for: 1997
Parsing: 1997-January.txt.gz
...
1997 had 3 mails
...
Parsing mails for: 1999
Parsing: 1999-January.txt.gz
Parsing: 1999-February.txt.gz
...
1999 had 803 mails
...
2009 had 7906 mails
...
73445 files in 215 seconds (341.6 files/second)
```

这会花几分钟时间，产生17MB的数据。你只需要做一次。

27.2.2　寻找最像给定文件的邮件

现在已经准备好搜索数据了，查询项是*我正在编写的文件*。开发这个程序时，我正在编写一个名为`sherlock_tfidf`的模块。我很想知道这个搜索引擎是否能正常工作，于是我输入了以下查询命令：

```
1> sherlock:find_mails_similar_to_file("2009", "./src/sherlock_tfidf.erl").
** searching for a mail in 2009 similar to the file:./src/sherlock_tfidf.erl
Searching for=[<<"idf">>,<<"word">>,<<"remove">>,<<"words">>,<<"tab">>,
<<"duplicates">>,<<"ets">>,<<"keywords">>,<<"bin">>,<<"skip">>,
<<"file">>,<<"index">>,<<"binary">>,<<"frequency">>,<<"dict">>]
7260 : 0.27 Word Frequency Analysis
7252 : 0.27 Word Frequency Analysis
7651 : 0.18 tab completion and word killing in the shell
4297 : 0.17 ets vs process-based registry + local vs global dispatch
5324 : 0.16 ets memory usage
5325 : 0.15 ets memory usage
1917 : 0.14 A couple of design questions
1860 : 0.12 leex and yecc spotting double newline
5361 : 0.11 dict slower than ets?
```

```
1991 : 0.11 Extending term external format to support   shared substructures
[7260,7252,7651,4297,5324,5325,1917,1860,5361,1991]
```

这个查询命令让sherlock在2009年的邮件里搜索与`sherlock_tfidf.erl`文件的内容相似的邮件。

输出的结果很有意思。首先，sherlock认为最能描述`sherlock_tfidf.erl`的是`idf`和`word`这些关键词，它们被列在了“searching for”（搜索项）这一行。随后，sherlock在所有发表于2009年的邮件里搜索这些关键词。它的输出如下：

```
7260 : 0.27 Word Frequency Analysis
7252 : 0.27 Word Frequency Analysis
7651 : 0.18 tab completion and word killing in the shell
4297 : 0.17 ets vs process-based registry + local vs global dispatch
...
```

每一行都有一个邮件索引编号，然后是相似度权重和标题行。相似度权重是一个0到1之间的数字，1意味着文档非常相似，0则是毫无相似性。得分最高的邮件是编号为7260的邮件，它的相似度权重是0.27。看上去有点意思，我们来了解一下它的详细内容。

```
2> sherlock:print_mail("2009", 7260).
----
ID: 7260
Date: Fri, 04 Dec 2009 17:57:03 +0100
From: ...
Subject: Word Frequency Analysis
Hello!

I need to compute a word frequency analysis of a fairly large corpus. At
present I discovered the disco database
http://discoproject.org/

which seems to include a tf-idf indexer. What about couchdb? I found an
article that it fails rather quickly (somewhere between 100 and 1000
wikipedia text pages)
...
```

当我第一次运行它时，结果让我非常激动。系统发现了一封讨论TF*IDF指数的邮件，而我并没有特意让它寻找TF*IDF指数，我只是说“去寻找与文件内部代码相似的所有邮件”。

现在我知道邮件7260颇有看头。或许还有别的邮件与这封邮件相似，我们来问问sherlock。

```
3> sherlock:find_mails_similar_to_mail("2009", "2009", 7260).
Searching for a mail in 2009 similar to mail number 7260 in 2009
Searching for=[<<"indexer">>,<<"analysis">>,<<"couchdb">>,<<"wortd">>,
<<"idf">>,<<"knuthellan.com">>,<<"frequncy">>,<<"corpus">>,
<<"dbm">>,<<"discoproject.org">>,<<"disco">>]
7252 : 0.84 Word Frequency Analysis
6844 : 0.21 couchdb in Karmic Koala
6848 : 0.21 couchdb in Karmic Koala
```

```
6847 : 0.20 couchdb in Karmic Koala
6849 : 0.19 couchdb in Karmic Koala
7264 : 0.17 Re: erlang search engine library?
6843 : 0.16 couchdb in Karmic Koala
2895 : 0.15 CouchDB integration
69 : 0.14 dialyzer fails when using packages and -r
[7252,6844,6848,6847,6849,7264,6843,2895,69]
```

这一次我搜索的是类似7260的邮件。邮件7252有着最高的相似度得分，而且的确与7260很相似，但是最终邮件7264更让我感兴趣。光凭相似度得分不足以判定哪些文件是我们想要的，系统只是提供了可能相似的文件。我们必须查看结果然后选择认为最有意思的邮件。

从`sherlock_tfidf.erl`的代码入手，我发现disco数据库有一个TF*IDF指数工具，几次查询后我又找到了与couchdb的联系。sherlock能找到各种有意思的东西供我进一步分析。

27.2.3 搜索指定作者、日期或标题的邮件

还可以执行所谓的单项（faceted）数据搜索。文档的单项是指用户名或标题之类的字段，单项搜索[1]是指限于特定字段或字段组的搜索。解析后的文档用Erlang记录表示，我们可以对所有文档执行任意字段的指定搜索。这里有一个单项搜索的例子：

```
1> sherlock:search_mails_regexprs("2009", "*Armstrong*", "*Protocol*", "*").
946:   UBF and JSON Protocols
5994:  Message protocol vs. Function call API
Query took:23 ms #results=2
[946,5994]
```

它搜索了满足以下条件的2009年邮件：作者名匹配正则表达式`*Armstrong*`，标题行匹配`*Protocol*`，内容不限。最终找到了两封匹配邮件：946和5994。可以像下面这样检查第一封邮件：

```
2> sherlock:print_mail("2009", 946).
----
ID: 946
Date: Sun, 15 Feb 2009 12:39:10 +0100
From: Joe Armstrong
Subject: UBF and JSON Protocols
For a long time I have been interested in describing protocols. In
2002 I published a contract system called UBF for defining protocols.
...
```

第二封也是如此。在这个阶段，可以执行更多查询，搜索特定数据或者寻找与前面这些邮件相似的邮件。

看了sherlock的演示之后，再来看看它的实现方式。

[1] http://en.wikipedia.org/wiki/Faceted_search

27.3　数据分区的重要性

告诉你一个秘密，给数据分区是让程序并行的关键。用来构建数据存储的process_all_years()是这样定义的：

```
process_all_years() ->
    [process_year(I) || I <- mail_years()].
```

process_year(Year)会处理某一年份的所有数据，而mail_years/0会返回一个年份列表。

要让程序并行，只需修改process_all_years的定义并调用pmap（在26.3节里讨论过）。做了这些小改动之后，我们的函数看起来就像下面这样：

```
process_all_years() ->
    lib_misc:pmap(fun(I) -> process_year(I) end, mail_years()).
```

这么做还有一个额外的不太明显的好处。要测试程序能否工作，只需要处理其中一个年份。如果有一台17+核的高端机器，就可以并行计算所有的年份。因为总共有17年的数据，所以需要至少17核的CPU以及允许同时进行17个输入操作的磁盘控制器。现代的固态硬盘有多个磁盘控制器，但在创作本书时使用的机器只有传统的硬盘和双核处理器，所以让程序并行化恐怕并不能显著提升速度。

我一直通过分析2009年的数据来测试这个程序。如果想加速程序，就需要使用高端机器，而我刚刚把顶层的列表推导修改为pmap来使程序并行化。

这种并行化方式正是映射–化简架构所使用的。为了快速搜索邮件数据，我们会使用17台机器，让每一台机器分别搜索一年的数据。把同一个查询命令发往全部17台机器，然后收集结果。这就是map-reduce的精髓，26.5节对此进行了讨论。

你会注意到这一章里的所有示例都使用2009作为基准年份。2009年有足够多的邮件来操练所有软件（共计7 906封邮件），而处理这些数量的邮件也不会花费太长时间，这一点在开发软件时很重要，因为程序需要修改并运行很多次。

因为我把数据按年份组织成独立的集合，所以只需要编写针对其中某一年的代码。如果有必要的话，我的程序稍作修改就能在更强力的机器上运行。

27.4　给邮件添加关键词

观察Erlang邮件列表里的邮件，就会发现它们不带任何关键词。但即使有关键词，还是会出现一个更深层次的问题。阅读同一份文档的两个人可能会对应该用哪些关键词来描述文档产生争议。因此，当我执行关键词搜索时，如果文档作者选择的关键词与我选择的不同，搜索就不灵了。

sherlock会为邮件列表里的每一封邮件计算关键词向量。我们可以问问它从2009年的邮件946里得出了什么关键词向量。

```
1> sherlock:get_keyword_vector("2009", 946).
[{"protocols",0.6983839995773734},
{"json",0.44660371850799946},
{"ubf",0.38889626945542854},
{"widely",0.2507841072279312},
{"1.html",0.17649029959852883},
{"recast",0.17130091468424488},
...
```

关键词向量是一个列表，包含得出的关键词和该词在文档里的重要性。词汇在文档里的重要性其实是该词在文档里的TF*IDF[①]权重。它是一个介于0和1之间的数字，0代表不重要，1代表非常重要。

为了计算两个文档的相似度，我们会分别计算它们的关键词向量，然后计算这些关键词向量的归一化向量积（normalized cross product），它被称为文档的*余弦相似度*[②]。如果两个文档的关键词有重叠，它们就存在一定的相似度，余弦相似度就是衡量这一点的。

现在来看看算法的细节。

27.4.1　词汇的重要性：TF*IDF权重

关键词重要性的一个常用量度就是所谓的TF*IDF权重。TF代表term frequency（词汇频度），IDF代表inverse document frequency（逆文档频度）。许多搜索引擎使用词汇在文档里的TF*IDF权重来评价词汇的重要性和寻找集合里的类似文档。在这一节里，我们将来看看TF*IDF权重是如何计算的。

在描述搜索方法的资料里，你会发现*语料库*（corpus）这个词经常出现。语料库是一个大型的参考文档集。在本书的案例中，语料库是Erlang邮件列表里73 445封邮件的集合。

计算TF*IDF权重要做的第一件事就是把感兴趣的文档分解成词汇序列。我们先规定词汇是由非字母字符分隔的字母字符串，现在假设在某个文档里找到了“socket”这个词，它是否重要取决于上下文。需要分析大量文档才能评估某个词的重要性。

假设单词“socket”在1%的语料库文档里出现。要计算它，可以分析语料库里的所有文档，统计有多少文档包含这个词。

如果来看单独的文档，它也可能会包含单词“socket”。一个词在文档里出现的次数除以文档的总词汇数被称为这个词的*词汇频度*。因此，如果“socket”在某个文档里出现了5次，而该文档共有100个词，那么该词的词汇频度就是5%。如果“socket”在语料库里的频度是1%，那么该文档的5%就显得非常重要，我们也许就应该选择“socket”作为关联文档的其中一个关键词。如果文档的词汇频度是1%，那么就和父语料库的出现概率相同，因此不具有重要性。

某个词在文档里的*词汇频度*（TF）就是它在文档里出现的次数除以文档的总词汇数。

单词`W`的*逆文档频度*（IDF）被定义为`log(Tot/N+1)`，其中`Tot`是语料库的总文档数，`N`是包

27

① http://en.wikipedia.org/wiki/Tf-idf

② http://en.wikipedia.org/wiki/Cosine_similarity

含单词W的文档数。

举个例子，假设有一个包含1000个文档的语料库。如果“orange”这个词出现在25个文档里，那么它的IDF就是`log(1000/26)`（= 1.58）。如果“orange”在一个100个词的文档里出现了10次，那么它的TF就是`10/100`（= 0.1），所以这个词的TF*IDF权重是0.158。

为了寻找某个文档的关键词集合，我们会计算文档里每个词的TF*IDF权重，然后采用那些TF*IDF权重最高的词。那些TF*IDF权重极低的词则会被忽略。

介绍完预备知识之后，现在可以来计算文档里的哪些词算是好关键词了。这是一个两次遍历的过程：第一次遍历会计算语料库里各个词的IDF，第二次遍历会计算出语料库里各个文档的关键词。

27.4.2　余弦相似度：两个权重向量的相似程度

用一个简单的示例来计算两个给定关键词向量的余弦相似度。假设有两个关键词向量：K1包含关键词a、b和c；K2包含关键词a、b和d。这些关键词和它们的TF*IDF权重如下：

```
1> K1 = [{a,0.5},{b,0.1},{c,0.2}].
[{a,0.5},{b,0.1},{c,0.2}]
2> K2 = [{a,0.3},{b,0.2},{d,0.6}].
[{a,0.3},{b,0.2},{d,0.6}]
```

K1和K2的向量积是关键词相同项的权重乘积之和。

```
3> Cross = 0.5*0.3 + 0.1*0.2.
0.16999999999999998.
```

为了计算余弦相似度，我们把向量积除以各个向量的范数（norm）。向量范数是各个权重平方之和的平方根。

```
4> Norm1 = math:sqrt(0.5*0.5 + 0.1*0.1 + 0.2*0.2).
0.5477225575051662
Norm2 = math:sqrt(0.3*0.3 + 0.2*0.2 + 0.6*0.6).
0.7
```

余弦相似度就是归一化向量积。

```
5> Cross/(Norm1*Norm2).
0.4433944513137058
```

它已被编写成一个库函数。

```
6> sherlock_similar:cosine_similarity(K1, K2).
0.4433944513137058
```

两个关键词向量的余弦相似度是一个介于0和1之间的数字。1代表这两个向量相同，0代表它们毫无相似性。

27.4.3　相似度查询

我们已经解释了相似度查询的所有预备知识。首先计算语料库里所有词的IDF，然后计算语料库里各个文档的关键词相似度向量。这些计算可能会花很长时间，但没有关系，因为它们只需要做一次。

要进行相似度查询，我们会取得待查询文档，并用语料库的IDF计算它的关键词向量。然后计算语料库里每个文档的余弦相似度，并选择那些具有最大余弦相似度系数的值。

我们会把结果列出来，让用户选择他们最感兴趣的文档。他们可以检查这些文档，也可以根据之前的分析结果进一步搜索其他文档。

27.5　实现方式概览

sherlock的所有代码都保存在本书主页代码库[1]的`code/sherlock`目录里。这一节从更高的层面概述了它的实现方式。处理过程的主要阶段如下。

- *初始化数据存储*

`sherlock_mail:ensure_mail_root/0`在一开始就被调用。这个方法能确保目录`${HOME}/.sherlock`存在。我们将把所有数据都保存在名为`${HOME}/.sherlock/mails`的目录里，在下文中用`MAIL`来指代它。

- *获取邮件索引*

第一步是获取`http://erlang.org/pipermail/erlang-questions/`上的数据，这是通过`inets`的HTTP客户端完成的，它是Erlang分发套装的一部分。`sherlock_get_mails:get_index/1`负责获取邮件，`sherlock_get_mails:parse_index/2`负责解析邮件。从服务器取回的HTML文件被保存为`MAIL/questions.html`，解析结果则保存在`MAIL/questions.term`文件里。

- *获取原始数据*

`sherlock_mails:fetch_all_mails/0`负责获取所有邮件。它一开始会读取`MAIL/questions.term`，然后获取所有压缩后的邮件文件，并把它们保存在`MAIL/cache`目录里。

- *给数据分区*

`sherlock_mails:find_mail_years/0`负责分析邮件缓存里的文件，然后返回一个已取回邮件的年份列表。

- *处理给定年份的数据*

`sherlock_mails:process_year(Year)`会做三件事：解析给定年份的所有数据，计算那一年所有邮件的TF*IDF权重，以及给各封邮件添加合成关键词。

下列数据文件会被创建。

- ❑ `MAIL/Year/parsed.bin`包含一个二进制型`B`，而`binary_to_term(B)`是一个由`#post`记

① http://pragprog.com/titles/jaerlang2/source_code

录组成的列表。

- MAIL/Year/idf.ets是一个保存若干{Word,Index,Idf}元组的ETS表，其中Word是一个二进制型（邮件里的某个词汇），Index是一个整数索引编号，而Idf是该词汇的IDF权重。
- MAIL/Year/mails.bin包含一个二进制型B，而binary_to_term(B)是一个由#post记录组成的列表。这一次记录里被加上了合成关键词。
- MAIL/Year/mails.list是一个包含mails.bin前几项的清单，开发程序时会用到它。偶尔可以用它检查输出，看看结果是否符合预期。

这一步的工作会分几处完成。sherlock_tfidf.erl负责计算TF*IDF权重，sherlock_mails:parse_mails/1负责解析压缩后的邮件文件，text_analyzers:standard_analyzer_factory/2负责把一个二进制型转换成词汇列表。

- 执行相似度查询

大多数工作都是由sherlock_mails:find_mails_similar_to_binary/2完成的。它负责读取MAIL/Year/idf.ets，并用它计算二进制型的关键词向量（通过调用sherlock_tfidf:keywords_in_binary/2完成）。随后，它对MAIL/Year/mails.bin里的所有条目进行迭代，而每一个条目都包含一个关键词向量。sherlock_similar:cosine_ similarity/2负责计算这些关键词向量的相似度，sherlock_best.erl则会保存一个包含最重要邮件的列表。

- 执行单项搜索

sherlock_mail:search_mails_regexp/4负责对MAIL/Year/mails.bin里的所有条目进行迭代，并对#post记录里的特定元素执行正则表达式搜索。

27.6　练习

sherlock的后续开发会在GitHub里进行[①]。如果能提供下列任何问题的解决方案，请创建一个sherlock分支，然后给我发送一个拉（pull）请求。正如你在这个清单里所见，sherlock程序有不少可以改进的地方，还有很多事可以做。

1. 找出模块的相似度

sherlock能找出Erlang邮件列表里各个邮件之间的相似度。添加一种功能来比较某个大型Erlang模块集合的相似度。

2. 找出模块历史

除了定义模块之间的相似度，还可以定义两个模块之间的*距离*。找到一组相似模块之后，请试着追溯模块的历史。这些模块是否是通过相互剪切粘贴代码派生出来的？如果找到了一组相似模块，能否追溯到它们的共同祖先？

3. 分析其他邮件列表的数据

sherlock只能提取来自pipermail存档的数据。编写一些能从其他常用邮件列表或论坛程序里提取邮件的代码。

① https://github.com/joearms/sherlock

4. Beautiful Soup

Python里有一个名为Beautiful Soup[①]的程序，它可以用来编写屏幕抓取器（screen-scraper）。实现一个Erlang版的Beautiful Soup，用它来收集某些流行邮件列表里的邮件。

5. 改进文本分析

改进文本分析器，看一看它生成的词汇。给Erlang编写一些自定义词汇生成器，用它们提取不同类型文件里的文本。

6. 单项搜索

给数据添加额外的字段并改进单项搜索。

7. 制作一个Web界面

给sherlock制作一个Web界面。

8. 给邮件打分

尝试给邮件打分，打分标准可以是谁写了邮件，谁评论了邮件，以及邮件产生了多少回复。

9. 制作结果图表

给某次查询计算出的相似度关系制作图表。在浏览器里展示这些图形。

10. 把邮件集转换成电子书

用epub格式生成。

27.7 总结

感谢你阅读本书，希望你能喜欢它。这是一段漫长的旅程，你学到了很多新东西。Erlang的世界观与其他编程语言很不一样，最大的区别在于如何处理错误以及如何实现并发。

并发是Erlang的本能。现实世界里的确存在通过消息交流的事物。我以前是一名物理学家，通过接收消息来感知世界。每一段光和声音都承载着信息。我们对世界的全部认识都来自于接收到的消息。

我们没有共享的记忆。我有我的记忆，你有你的记忆，我不知道你在想些什么。如果我想知道你的想法，就必须问你一个问题并等待答复。

Erlang使用的编程模型和这个世界的运作方式非常相似。这让编程变得简单。许多程序员发现了这一点，许多公司也是。

那么，接下来该做什么？把Erlang告诉别人，加入Erlang邮件列表，参加世界各地举办的Erlang会议，解决一些有意思的问题，并且动手让世界变得更美好。

① http://www.crummy.com/software/BeautifulSoup/

附录 A

OTP模板

本附录包含gen_serverand、supervisor和application的完整模板清单。这些模板内建于Emacs模式。

A.1 通用服务器模板

gen_server_template.full

```
%%%-------------------------------------------------------------------
%%% @作者XXX<me@hostname.local>
%%% @版权所有 (C) 2013, XXX
%%% @doc
%%%
%%% @end
%%% 创建于：2013年5月26日 作者XXX <me@hostname.local>
%%%-------------------------------------------------------------------
-module().

-behaviour(gen_server).

%% API
-export([start_link/0]).

%% gen_server回调函数
-export([init/1, handle_call/3, handle_cast/2, handle_info/2,
         terminate/2, code_change/3]).

-define(SERVER, ?MODULE).

-record(state, {}).

%%%===================================================================
%%% API
%%%===================================================================

%%--------------------------------------------------------------------
%% @doc
%% 启动服务器
```

```
%%
%% @spec start_link() -> {ok, Pid} | ignore | {error, Error}
%% @end
%%--------------------------------------------------------------------
start_link() ->
    gen_server:start_link({local, ?SERVER}, ?MODULE, [], []).

%%%===================================================================
%%% gen_server回调函数
%%%===================================================================

%%--------------------------------------------------------------------
%% @private
%% @doc
%% 初始化服务器
%%
%% @spec init(Args) -> {ok, State} |
%%                     {ok, State, Timeout} |
%%                     ignore |
%%                     {stop, Reason}
%% @end
%%--------------------------------------------------------------------
init([]) ->
    {ok, #state{}}.

%%--------------------------------------------------------------------
%% @private
%% @doc
%% 处理调用消息
%%
%% @spec handle_call(Request, From, State) ->
%%                                   {reply, Reply, State} |
%%                                   {reply, Reply, State, Timeout} |
%%                                   {noreply, State} |
%%                                   {noreply, State, Timeout} |
%%                                   {stop, Reason, Reply, State} |
%%                                   {stop, Reason, State}
%% @end
%%--------------------------------------------------------------------
handle_call(_Request, _From, State) ->
    Reply = ok,
    {reply, Reply, State}.

%%--------------------------------------------------------------------
%% @private
%% @doc
%% 处理播发消息
%%
%% @spec handle_cast(Msg, State) -> {noreply, State} |
%%                                  {noreply, State, Timeout} |
%%                                  {stop, Reason, State}
%% @end
%%--------------------------------------------------------------------
```

```
handle_cast(_Msg, State) ->
    {noreply, State}.

%%--------------------------------------------------------------------
%% @private
%% @doc
%% 处理所有非调用/播发的消息
%%
%% @spec handle_info(Info, State) -> {noreply, State} |
%%                                   {noreply, State, Timeout} |
%%                                   {stop, Reason, State}
%% @end
%%--------------------------------------------------------------------
handle_info(_Info, State) ->
    {noreply, State}.

%%--------------------------------------------------------------------
%% @private
%% @doc
%% 这个函数是在某个gen_server即将终止时调用的。它应当是Module:init/1的逆操作，并进行必要的清理。
%% 当它返回时，gen_server终止并生成原因Reason。它的返回值会被忽略
%%
%% @spec terminate(Reason, State) -> void()
%% @end
%%--------------------------------------------------------------------
terminate(_Reason, _State) ->
    ok.

%%--------------------------------------------------------------------
%% @private
%% @doc
%% 在代码更改时转换进程状态
%%
%% @spec code_change(OldVsn, State, Extra) -> {ok, NewState}
%% @end
%%--------------------------------------------------------------------
code_change(_OldVsn, State, _Extra) ->
    {ok, State}.

%%%===================================================================
%%% 内部函数
%%%===================================================================
```

A.2 监控器模板

supervisor_template.full

```
%%%-------------------------------------------------------------------
%%% @作者XXX <me@hostname.local>
%%% @版权所有 (C) 2013, XXX
%%% @doc
```

```
%%%
%%% @end
%%% 创建于：2013年5月26日 作者XXX <me@hostname.local>
%%%-------------------------------------------------------------------
-module().

-behaviour(supervisor).
%% API
-export([start_link/0]).

%% 监控器回调函数
-export([init/1]).

-define(SERVER, ?MODULE).

%%%===================================================================
%%% API函数
%%%===================================================================

%%--------------------------------------------------------------------
%% @doc
%% 启动监控器
%%
%% @spec start_link() -> {ok, Pid} | ignore | {error, Error}
%% @end
%%--------------------------------------------------------------------
start_link() ->
    supervisor:start_link({local, ?SERVER}, ?MODULE, []).

%%%===================================================================
%%% 监控器回调函数
%%%===================================================================
%%--------------------------------------------------------------------
%% @private
%% @doc
%% 每当supervisor:start_link/[2,3]启动一个监控器时，新进程就会调用这个函数来确定重启策略、
%% 最大重启频率和子进程规范
%% specifications.
%%
%% @spec init(Args) -> {ok, {SupFlags, [ChildSpec]}} |
%% ignore |
%% {error, Reason}
%% @end
%%--------------------------------------------------------------------
init([]) ->
    RestartStrategy = one_for_one,
    MaxRestarts = 1000,
    MaxSecondsBetweenRestarts = 3600,

    SupFlags = {RestartStrategy, MaxRestarts, MaxSecondsBetweenRestarts},

    Restart = permanent,
    Shutdown = 2000,
```

```
    Type = worker,

    AChild = {'AName', {'AModule', start_link, []},
              Restart, Shutdown, Type, ['AModule']},

    {ok, {SupFlags, [AChild]}}.

%%%===================================================================
%%% 内部函数
%%%===================================================================
```

A.3　应用程序模板

application_template.full

```
%%%-------------------------------------------------------------------
%%% @作者XXX <me@hostname.local>
%%% @版权所有 (C) 2013, XXX
%%% @doc
%%%
%%% @end
%%% 创建于：2013年5月26日 作者XXX <me@hostname.local>
%%%-------------------------------------------------------------------
-module().

-behaviour(application).

%% 应用程序回调函数
-export([start/2, stop/1]).

%%%===================================================================
%%% 应用程序回调函数
%%%===================================================================

%%--------------------------------------------------------------------
%% @private
%% @doc
%% 用application:start/[1,2]启动一个应用程序时会调用这个函数。它应当启动该应用程序的各个进程。
%% 如果应用程序的结构遵循OTP的监控树设计原则，此函数就会启动该树的顶级监控进程。
%%
%% @spec start(StartType, StartArgs) -> {ok, Pid} |
%%                                      {ok, Pid, State} |
%%                                      {error, Reason}
%%      StartType = normal | {takeover, Node} | {failover, Node}
%%      StartArgs = term()
%% @end
%%--------------------------------------------------------------------
start(_StartType, _StartArgs) ->
    case 'TopSupervisor':start_link() of
        {ok, Pid} ->
            {ok, Pid};
```

```
        Error ->
            Error
    end.

%%----------------------------------------------------------------------
%% @private
%% @doc
%% 这个函数是在应用程序停止时调用的。它应当是Module:start/2的逆操作，并进行必要的清理。它的返回
%% 值会被忽略。
%%
%% @spec stop(State) -> void()
%% @end
%%----------------------------------------------------------------------
stop(_State) ->
    ok.

%%%======================================================================
%%% 内部函数
%%%======================================================================
```

附录 B

一个套接字应用程序

本附录是关于实现lib_chan库的，我们在14.6.1节里介绍过它。lib_chan的代码在TCP/IP之上实现了一个完整的网络层，能够提供认证和Erlang数据流功能。一旦理解了lib_chan的原理，就能量身定制我们自己的通信基础结构，并把它叠加在TCP/IP之上了。

就lib_chan本身而言，它是一种构建分布式系统的有用组件。

为了保持本附录的完整性，以下内容会与14.6.1节里的材料出现部分重复。

本附录里的代码是我介绍过的最复杂的代码之一，所以如果第一次阅读时不能理解也别担心。如果你只想使用lib_chan而不关心它是如何实现的，阅读第一节就够了，其余部分可以跳过。

B.1 一个示例

首先，用一个简单的示例来展示如何使用lib_chan。我们会创建一个简单的服务器，让它计算阶乘和斐波那契数，并用一个密码来保护它。

这个服务器将在2233端口工作。

创建服务器的过程共分四步。

(1) 编写配置文件。

(2) 编写服务器代码。

(3) 启动服务器。

(4) 通过网络访问服务器。

B.1.1 步骤 1：编写配置文件

以下是这个示例的配置文件：

socket_dist/config1

```
{port, 2233}.
{service, math, password, "qwerty", mfa, mod_math, run, []}.
```

这个配置文件里有一些service元组，它们的形式如下：

```
{service, <Name>, password, <P>, mfa, <Mod>, <Func>, <ArgList>}
```

里面的参数由原子service、password和mfa分隔。mfa是“module, function, args”的缩写，意思是接下来的三个参数应当被解释为模块名、函数名和一个用来调用函数的参数列表。

在我们的示例里，配置文件指定了一个名为math（数学）的服务，它的工作端口是2233。这个服务由密码qwerty保护，实现它的模块名为mod_math，启动方式是调用mod_math:run/3，run/3的第三个参数是[]。

B.1.2　步骤 2：编写服务器代码

这个数学服务器的代码如下：

socket_dist/mod_math.erl

```
-module(mod_math).
-export([run/3]).
run(MM, ArgC, ArgS) ->
    io:format("mod_math:run starting~n"
              "ArgC = ~p ArgS=~p~n",[ArgC, ArgS]),
    loop(MM).
loop(MM) ->
    receive
        {chan, MM, {factorial, N}} ->
            MM ! {send, fac(N)},
            loop(MM);
        {chan, MM, {fibonacci, N}} ->
            MM ! {send, fib(N)},
            loop(MM);
        {chan_closed, MM} ->
            io:format("mod_math stopping~n"),
            exit(normal)
    end.
fac(0) -> 1;
fac(N) -> N*fac(N-1).
fib(1) -> 1;
fib(2) -> 1;
fib(N) -> fib(N-1) + fib(N-2).
```

当某个客户端连接到2233端口并请求math服务时，lib_auth会对它进行认证，如果密码正确，就会通过mod_math:run(MM, ArgC, ArgS)函数分裂出一个处理进程。MM是中间人的PID，ArgC来自客户端，ArgS则来自配置文件。

如果客户端向服务器发送一个消息X，到达后就会转换成消息{chan, MM, X}。如果客户端挂了或者连接出现问题，服务器就会收到一个{chan_closed, MM}消息。要向客户端发送消息Y，服务器需要执行MM ! {send, Y}，要关闭通信信道就执行MM ! close。

这个数学服务器很简单，它所做的就是等待一个{chan, MM, {factorial, N}}消息，然后执行MM ! {send, fac(N)}来把结果发回客户端。

B.1.3　步骤 3：启动服务器

像下面这样启动服务器：

```
1>  lib_chan:start_server("./config1").
ConfigData=[{port,2233},{service,math,password,"qwerty",mfa,mod_math,run,[]}]
true
```

B.1.4　步骤 4：通过网络访问服务器

可以在单台机器上进行代码测试。

```
2> {ok, S} = lib_chan:connect("localhost",2233,math,
                              "qwerty",{yes,go}).

{ok,<0.47.0>}
3> lib_chan:rpc(S, {factorial,20}).
2432902008176640000
4> lib_chan:rpc(S, {fibonacci,15}).
610
4> lib_chan:disconnect(S).
close
```

B.2　lib_chan是如何工作的

构建lib_chan使用了四个模块里的代码。

- lib_chan扮演“主模块”的角色。程序员只需要了解lib_chan所导出的那些方法。其他三个模块（稍后讨论）会在lib_chan的内部使用。
- lib_chan_mm负责编码和解码Erlang消息，并管理套接字通信。
- lib_chan_cs负责设立服务器并管理客户端连接。它的主要工作之一是限制同时连接的最大客户端数量。
- lib_chan_auth包含的代码用于进行简单的质询/响应认证。

B.2.1　lib_chan

lib_chan的结构如下。

```
-module(lib_chan).

start_server(ConfigFile) ->
    %% 读取配置文件并检查语法
    %% 调用start_port_server(Port, ConfigData)
    %% 其中Port是所需的端口，ConfigData包含配置数据
```

```
start_port_server(Port, ConfigData) ->
   lib_chan_cs:start_raw_server( ..
          fun(Socket) ->
              start_port_instance(Socket, ConfigData),
          end, ... )
   %% lib_chan_cs负责管理连接。
   %% 新连接建立后会调用start_raw_server的参数，
   %% 也就是这个fun。
start_port_instance(Socket, ConfigData) ->
   %% 它会在客户端连接服务器时执行分裂。
   %% 我们会设立一个中间人并执行认证，
   %% 如果一切顺利就调用
   %% really_start(MM, ArgC, {Mod, Func, ArgS})
   %%（后三个参数来自配置文件）

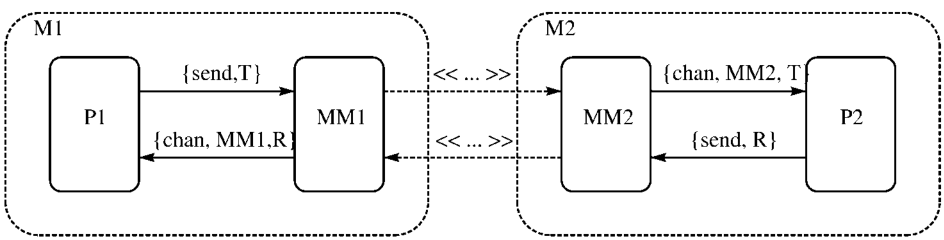
really_start(MM, ArgC, {Mod, Func, ArgS}) ->
   apply(Mod, Func, [MM, ArgC, ArgS]).

connect(Host, Port, Service, Password, ArgC) ->
   %% 客户端代码
```

B.2.2　lib_chan_mm：中间人

lib_chan_mm实现了一个中间人。它能对应用程序隐藏套接字通信，并把TCP套接字上的数据流转变成Erlang消息。中间人负责组装消息（它可能是碎片化的）和编码/解码Erlang数据类型，也就是把它们转换成能通过套接字发送和接收的字节流。

现在是看一下图B-1的好时机，它展示了我们的中间人架构。当M1机器上的P1进程想给M2机器上的P2进程发送消息T时，它就会执行MM1 ! {send, T}。MM1扮演P2的代理角色。任何发给MM1的消息都会被编码、写入套接字，然后发给MM2。MM2会解码从套接字收到的所有数据，然后把消息{chan, MM2, T}发给P2。



图B-1　带中间人的套接字通信

M1机器上的MM1进程表现得就像是P2的代理，而在M2机器上的MM2进程表现得就像是P1的代理。

MM1和MM2都是中间人进程的PID。中间人进程的代码如下。

```
loop(Socket, Pid) ->
    receive
        {tcp, Socket, Bin} ->
            Pid ! {chan, self(), binary_to_term(Bin)},
            loop(Socket, Pid);
        {tcp_closed, Socket} ->
            Pid ! {chan_closed, self()};
        close ->
            gen_tcp:close(Socket);
        {send, T} ->
            gen_tcp:send(Socket, [term_to_binary(T)]),
            loop(Socket, Pid)
    end.
```

这个循环是套接字数据和Erlang消息传输这两个世界之间的接口。稍后（参阅B.3.3节）你会看到lib_chan_mm的完整代码，它比这里展示的代码略微复杂一些，但原理是相同的。唯一的区别是添加了跟踪消息的代码和一些接口方法。

B.2.3 lib_chan_cs

lib_chan_cs负责设立客户端和服务器通信。下面是它导出的两个重要方法。

- start_raw_server(Port, Max, Fun, PacketLength)
  它会启动一个监听器来监听Port上的连接。允许的最大同时会话数是Max。Fun是一个元数为1的fun，Fun(Socket)会在连接开始时执行。套接字通信会假定包长度为PacketLength。

- start:raw_client(Host, Port, PacketLength) => {ok, Socket} | {error, Why}
  它会尝试连接由start_raw_server打开的端口。

lib_chan_cs的代码遵循17.1.3节里描述的模式，此外它还限制同时打开的最大连接数。虽然这个小细节从概念上讲十分简单，但它需要增加20行左右的晦涩代码来执行捕捉退出之类的任务。这样的代码看上去一团糟，但是不必担心，它能很好地完成自己的工作，并对模块的用户隐藏其中的复杂性。

B.2.4 lib_chan_auth

这个模块实现了一种简单形式的质询/响应认证。质询/响应认证基于“关联服务名的共享秘密”这一概念。要展示它是如何工作的，我们将假设一个名为math的服务具有qwerty这个共享秘密。

如果某个客户端想使用math服务，就必须向服务器证明它知道共享秘密。这个过程如下所示。

(1) 客户端向服务器发送一个请求来表示它希望使用math服务。

(2) 服务器计算出一个随机字符串C，然后把它发给客户端。这就是质询。字符串是由lib_chan_auth:make_challenge()函数生成的。可以用交互方式来看它是如何工作的。

```
1> C = lib_chan_auth:make_challenge().
"qnyrgzqefvnjdombanrsmxikc"
```

(3) 客户端接收字符串（C）并计算出响应（R），其中R = MD5(C ++ Secret)，它是由lib_chan_auth:make_response生成的。这里有一个例子：

```
2> R = lib_chan_auth:make_response(C, "qwerty").
"e759ef3778228beae988d91a67253873"
```

(4) 这个响应被发回服务器。服务器接收响应并检查它是否正确，做法是算出预期的响应值。这是由lib_chan_auth:is_response_correct实现的。

```
3> lib_chan_auth:is_response_correct(C, R, "qwerty").
true
```

B.3　lib_chan代码

是时候来看看代码了。

B.3.1　lib_chan

socket_dist/lib_chan.erl

```
-module(lib_chan).
-export([cast/2, start_server/0, start_server/1,
         connect/5, disconnect/1, rpc/2]).
-import(lists, [map/2, member/2, foreach/2]).
-import(lib_chan_mm, [send/2, close/1]).

%%----------------------------------------------------------------------
%% 服务器代码

start_server() ->
    case os:getenv("HOME") of
        false ->
            exit({ebadEnv, "HOME"});
        Home ->
            start_server(Home ++ "/.erlang_config/lib_chan.conf")
    end.

start_server(ConfigFile) ->
    io:format("lib_chan starting:~p~n",[ConfigFile]),
    case file:consult(ConfigFile) of
        {ok, ConfigData} ->
            io:format("ConfigData=~p~n",[ConfigData]),
            case check_terms(ConfigData) of
                [] ->
```

```
                start_server1(ConfigData);
            Errors ->
                exit({eDaemonConfig, Errors})
        end;
    {error, Why} ->
        exit({eDaemonConfig, Why})
end.

%% check_terms() -> [Error]
check_terms(ConfigData) ->
    L = map(fun check_term/1, ConfigData),
    [X || {error, X} <- L].
check_term({port, P}) when is_integer(P)     -> ok;
check_term({service,_,password,_,mfa,_,_,_}) -> ok;
check_term(X) -> {error, {badTerm, X}}.

start_server1(ConfigData) ->
    register(lib_chan, spawn(fun() -> start_server2(ConfigData) end)).

start_server2(ConfigData) ->
    [Port] = [ P || {port,P} <- ConfigData],
    start_port_server(Port, ConfigData).

start_port_server(Port, ConfigData) ->
    lib_chan_cs:start_raw_server(Port,
                                 fun(Socket) ->
                                         start_port_instance(Socket,
                                                             ConfigData) end,
                                 100,
                                 4).

start_port_instance(Socket, ConfigData) ->
    %% 这是处理底层连接的位置
    %% 但首先要分裂出一个连接处理进程，必须成为中间人

    S = self(),
    Controller = spawn_link(fun() -> start_erl_port_server(S, ConfigData) end),
    lib_chan_mm:loop(Socket, Controller).

start_erl_port_server(MM, ConfigData) ->
    receive
        {chan, MM, {startService, Mod, ArgC}} ->
            case get_service_definition(Mod, ConfigData) of
                {yes, Pwd, MFA} ->
                    case Pwd of
                        none ->
                            send(MM, ack),
                            really_start(MM, ArgC, MFA);
                        _ ->
```

```
                    do_authentication(Pwd, MM, ArgC, MFA)
                end;
            no ->
                io:format("sending bad service~n"),
                send(MM, badService),
                close(MM)
        end;
    Any ->
        io:format("*** ErL port server got:~p ~p~n",[MM, Any]),
        exit({protocolViolation, Any})
    end.
do_authentication(Pwd, MM, ArgC, MFA) ->
    C = lib_chan_auth:make_challenge(),
    send(MM, {challenge, C}),
    receive
        {chan, MM, {response, R}} ->
            case lib_chan_auth:is_response_correct(C, R, Pwd) of
                true ->
                    send(MM, ack),
                    really_start(MM, ArgC, MFA);
                false ->
                    send(MM, authFail),
                    close(MM)
            end
    end.


%% MM是中间人
%% Mod是我们想要执行的模块。ArgC和ArgS分别来自客户端和服务器

really_start(MM, ArgC, {Mod, Func, ArgS}) ->
    %% 认证成功，现在开始工作
    case (catch apply(Mod,Func,[MM,ArgC,ArgS])) of
        {'EXIT', normal} ->
            true;
        {'EXIT', Why} ->
            io:format("server error:~p~n",[Why]);
        Why ->
            io:format("server error should die with exit(normal) was:~p~n",
                      [Why])
    end.

%% get_service_definition(Name, ConfigData)

get_service_definition(Mod, [{service, Mod, password, Pwd, mfa, M, F, A}|_]) ->
    {yes, Pwd, {M, F, A}};
get_service_definition(Name, [_|T]) ->
    get_service_definition(Name, T);
get_service_definition(_, []) ->
```

```
    no.

%%--------------------------------------------------------------------
%% 客户端连接代码
%% connect(...) -> {ok, MM} | Error

connect(Host, Port, Service, Secret, ArgC) ->
    S = self(),
    MM = spawn(fun() -> connect(S, Host, Port) end),
    receive
        {MM, ok} ->
            case authenticate(MM, Service, Secret, ArgC) of
                ok    -> {ok, MM};
                Error -> Error
            end;
        {MM, Error} ->
            Error
    end.

connect(Parent, Host, Port) ->
    case lib_chan_cs:start_raw_client(Host, Port, 4) of
        {ok, Socket} ->
            Parent ! {self(), ok},
            lib_chan_mm:loop(Socket, Parent);
        Error ->
            Parent ! {self(),  Error}
    end.

authenticate(MM, Service, Secret, ArgC) ->
    send(MM, {startService, Service, ArgC}),
    %% 应该会收到质询、ack或者套接字已关闭的消息
    receive
        {chan, MM, ack} ->
            ok;
        {chan, MM, {challenge, C}} ->
            R = lib_chan_auth:make_response(C, Secret),
            send(MM, {response, R}),
            receive
                {chan, MM, ack} ->
                    ok;
                {chan, MM, authFail} ->
                    wait_close(MM),
                    {error, authFail};
                Other ->
                    {error, Other}
            end;
        {chan, MM, badService} ->
            wait_close(MM),
            {error, badService};
        Other ->
```

```
            {error, Other}
    end.

wait_close(MM) ->
    receive
        {chan_closed, MM} ->
            true
    after 5000 ->
            io:format("**error lib_chan~n"),
            true
    end.

disconnect(MM) -> close(MM).

rpc(MM, Q) ->
    send(MM, Q),
    receive
        {chan, MM, Reply} ->
            Reply
    end.

cast(MM, Q) ->
    send(MM, Q).
```

B.3.2　lib_chan_cs

socket_dist/lib_chan_cs.erl

```
-module(lib_chan_cs).
%% cs代表client_server
-export([start_raw_server/4, start_raw_client/3]).
-export([stop/1]).
-export([children/1]).

%% start_raw_server(Port, Fun, Max, PacketLength)
%% 这个服务器在Port上接受最多Max个连接
%% 首次连接Port时，会调用Fun(Socket)
%% 此后发给套接字的消息会转换成发给处理进程的消息
%% PacketLength通常是0、1、2或4（详情见inet的手册页）                     ).

%% tcp_server的典型用法如下[1]：
%% 设立一个监听器
%%   start_agent(Port) ->
%%     process_flag(trap_exit, true),
%%     lib_chan_server:start_raw_server(Port,
%%                                      fun(Socket) -> input_handler(Socket) end,
%%                                      15, 0).

start_raw_client(Host, Port, PacketLength) ->
```

```
    gen_tcp:connect(Host, Port,
                    [binary, {active, true}, {packet, PacketLength}]).

%% 注意：当start_raw_server返回时
%% 它应该已经准备好立即接受请求了

start_raw_server(Port, Fun, Max, PacketLength) ->
    Name = port_name(Port),
    case whereis(Name) of
        undefined ->
            Self = self(),
            Pid = spawn_link(fun() ->
                                 cold_start(Self,Port,Fun,Max,PacketLength)
                             end),
            receive
                {Pid, ok} ->
                    register(Name, Pid),
                    {ok, self()};
                {Pid, Error} ->
                    Error
            end;
        _Pid ->
            {error, already_started}
    end.

stop(Port) when integer(Port) ->
    Name = port_name(Port),
    case whereis(Name) of
        undefined ->
            not_started;
        Pid ->
            exit(Pid, kill),
            (catch unregister(Name)),
            stopped
    end.
children(Port) when integer(Port) ->
    port_name(Port) ! {children, self()},
    receive
        {session_server, Reply} -> Reply
    end.


port_name(Port) when integer(Port) ->
    list_to_atom("portServer" ++ integer_to_list(Port)).


cold_start(Master, Port, Fun, Max, PacketLength) ->
    process_flag(trap_exit, true),
    %% 现在我们准备好运行了
```

```
    case gen_tcp:listen(Port, [binary,
                               %% {dontroute, true},
                               {nodelay,true},
                               {packet, PacketLength},
                               {reuseaddr, true},
                               {active, true}]) of
        {ok, Listen} ->
            %% io:format("Listening to:~p~n",[Listen]),
            Master ! {self(), ok},
            New = start_accept(Listen, Fun),
            %% Now we're ready to run
            socket_loop(Listen, New, [], Fun, Max);
        Error ->
            Master ! {self(), Error}
    end.

socket_loop(Listen, New, Active, Fun, Max) ->
    receive
        {istarted, New} ->
            Active1 = [New|Active],
            possibly_start_another(false,Listen,Active1,Fun,Max);
        {'EXIT', New, _Why} ->
            %% io:format("Child exit=~p~n",[Why]),
            possibly_start_another(false,Listen,Active,Fun,Max);
        {'EXIT', Pid, _Why} ->
            %% io:format("Child exit=~p~n",[Why]),
            Active1 = lists:delete(Pid, Active),
            possibly_start_another(New,Listen,Active1,Fun,Max);
        {children, From} ->
            From ! {session_server, Active},
            socket_loop(Listen,New,Active,Fun,Max);
        _Other ->
            socket_loop(Listen,New,Active,Fun,Max)
    end.

possibly_start_another(New, Listen, Active, Fun, Max)
  when pid(New) ->
    socket_loop(Listen, New, Active, Fun, Max);
possibly_start_another(false, Listen, Active, Fun, Max) ->
    case length(Active) of
        N when N < Max ->
            New = start_accept(Listen, Fun),
            socket_loop(Listen, New, Active, Fun,Max);
        _ ->
            socket_loop(Listen, false, Active, Fun, Max)
    end.

start_accept(Listen, Fun) ->
    S = self(),
    spawn_link(fun() -> start_child(S, Listen, Fun) end).
```

```
start_child(Parent, Listen, Fun) ->
    case gen_tcp:accept(Listen) of
        {ok, Socket} ->
            Parent ! {istarted,self()},             % 告知控制器
            inet:setopts(Socket, [{packet,4},
                                  binary,
                                  {nodelay,true},
                                  {active, true}]),
            %% 激活套接字之前
            %% io:format("running the child:~p Fun=~p~n", [Socket, Fun]),
            process_flag(trap_exit, true),
            case (catch Fun(Socket)) of
                {'EXIT', normal} ->
                    true;
                {'EXIT', Why} ->
                io:format("Port process dies with exit:~p~n",[Why]),
                true;
            _ ->
                %% 不是退出消息，说明一切顺利
                true
        end
end.
```

B.3.3　lib_chan_mm

socket_dist/lib_chan_mm.erl

```
%% 协议
%%   发往控制进程
%%      {chan, MM, Term}
%%      {chan_closed, MM}
%%   来自任意进程
%%      {send, Term}
%%      close

-module(lib_chan_mm).
%% TCP中间人
%% 模拟gen_tcp接口

-export([loop/2, send/2, close/1, controller/2, set_trace/2, trace_with_tag/2]).

send(Pid, Term)       -> Pid ! {send, Term}.
close(Pid)            -> Pid ! close.
controller(Pid, Pid1) -> Pid ! {setController, Pid1}.
set_trace(Pid, X)     -> Pid ! {trace, X}.
```

① 相关内容可参阅：http://erlang.org/pipermail/erlang-questions/attachments/20011009/824600c5/attachment.ksh。
——译者注

```
trace_with_tag(Pid, Tag) ->
    set_trace(Pid, {true,
                    fun(Msg) ->
                            io:format("MM:~p ~p~n",[Tag, Msg])
                    end}).

loop(Socket, Pid) ->
    %% trace_with_tag(self(), trace)
    process_flag(trap_exit, true),
    loop1(Socket, Pid, false).

loop1(Socket, Pid, Trace) ->
    receive
        {tcp, Socket, Bin} ->
            Term = binary_to_term(Bin),
            trace_it(Trace,{socketReceived, Term}),
            Pid ! {chan, self(), Term},
            loop1(Socket, Pid, Trace);
        {tcp_closed, Socket} ->
            trace_it(Trace, socketClosed),
            Pid ! {chan_closed, self()};
        {'EXIT', Pid, Why} ->
            trace_it(Trace,{controllingProcessExit, Why}),
            gen_tcp:close(Socket);
        {setController, Pid1} ->
            trace_it(Trace, {changedController, Pid}),
            loop1(Socket, Pid1, Trace);
        {trace, Trace1} ->
            trace_it(Trace, {setTrace, Trace1}),
            loop1(Socket, Pid, Trace1);
        close ->
            trace_it(Trace, closedByClient),
            gen_tcp:close(Socket);
        {send, Term}  ->
            trace_it(Trace, {sendingMessage, Term}),
            gen_tcp:send(Socket, term_to_binary(Term)),
            loop1(Socket, Pid, Trace);
        UUg ->
            io:format("lib_chan_mm: protocol error:~p~n",[UUg]),
            loop1(Socket, Pid, Trace)
    end.
trace_it(false, _)     -> void;
trace_it({true, F}, M) -> F(M).
```

B.3.4 lib_chan_auth

socket_dist/lib_chan_auth.erl

```
-module(lib_chan_auth).
-export([make_challenge/0, make_response/2, is_response_correct/3]).

make_challenge() ->
    random_string(25).
make_response(Challenge, Secret) ->
    lib_md5:string(Challenge ++ Secret).
is_response_correct(Challenge, Response, Secret) ->
    case lib_md5:string(Challenge ++ Secret) of
        Response -> true;
        _        -> false
    end.

%% random_string(N) -> 一个随机字符串（内含N个字符）
random_string(N) -> random_seed(), random_string(N, []).
random_string(0, D) -> D;
random_string(N, D) ->
    random_string(N-1, [random:uniform(26)-1+$a|D]).
random_seed() ->
    {_,_,X} = erlang:now(),
    {H,M,S} = time(),
    H1 = H * X rem 32767,
    M1 = M * X rem 32767,
    S1 = S * X rem 32767,
    put(random_seed, {H1,M1,S1}).
```

附录 C

一种简单的执行环境

在本附录里，我们将构建一种运行Erlang程序的简单执行环境（Simple Execution Environment，简称SEE）。之所以把代码放入本书的附录而不是正文是因为它的风格不同。本书里的所有代码都适合在标准的Erlang/OTP分发套装里运行，而这些代码会有意避免使用Erlang库代码，尽量只使用Erlang的基本函数。

第一次见到Erlang时，人们经常会分不清哪些属于语言，哪些属于操作环境。OTP提供了一种类似操作系统的丰富环境来长期运行分布式Erlang应用程序。然而，Erlang（语言）和OTP（环境）到底都提供了哪些功能却很难分辨。

SEE提供了一种“更接近本质”的环境，它能够更好地区分哪些是Erlang提供的，哪些是OTP提供的。SEE所提供的都包含在单个模块内。OTP启动时会载入60个左右的模块，很难一眼看出它的工作方式，但如果知道该往哪儿看，就不会特别复杂了。启动文件是第一个该看的。从启动文件入手然后阅读`init.erl`里的代码，一切就能了然于胸了。

SEE环境可以用于脚本编程，因为它的启动速度很快，也可以用于嵌入式开发，因为它非常小。要做到这些，你需要了解Erlang是如何启动的，以及代码自动载入系统是如何工作的。

当启动一个标准Erlang系统时（通过shell命令`erl`），会载入67个模块并启动25个进程，然后程序才能运行。这需要大概1秒钟的时间。如果想执行的程序不需要标准系统提供的这些好东西，时间就可以缩短到几十毫秒。

弄清这67个模块和进程的作用是个艰巨的任务，但还有一条更短的路可走。SEE简化了系统，使你只需研究一个模块就能理解代码载入系统与提供I/O服务的方式。SEE只用一个模块就能提供自动载入、通用服务器进程和错误处理功能。

启动SEE之前，先来收集一些OTP系统的统计信息以供将来参考。

```
$ erl
1> length([I||{I,X} <- code:all_loaded(), X =/= preloaded]).
67
2> length(processes()).
25
3> length(registered()).
16
```

`code:all_loaded()`会返回一个列表，内含系统当前载入的所有模块。`processes()`是一

个内置函数，它会返回一个包含所有系统已知进程的列表，而registered()会返回一个包含所有已注册进程的列表。

由此可见，光启动系统就会载入67个模块并启动25个进程，其中有16个注册进程。我们来看看能否减少这些数目。

C.1　Erlang是如何启动的

当Erlang启动时，它会读取一个启动文件并执行里面的命令。有8个Erlang模块是预先载入的，这些模块已被编译为C并链接到Erlang的虚拟机里。这8个模块负责启动系统，其中包括init.er（读取和执行启动文件里的命令）和erl_prim_loader（把代码载入系统）。

启动文件包含一个通过term_to_binary(Script)创建的二进制型，其中Script是一个包含启动脚本的元组。

我们将制作一个新的启动文件（名为see.boot）和一段脚本（名为see），后者会用这个新文件来启动程序。启动文件会载入少量模块，其中包括see.erl，它将包含我们的定制执行环境。启动文件和脚本的创建方式是执行make_scripts/0。

see/see.erl

```
make_scripts() ->
    {ok, Cwd} = file:get_cwd(),
    Script =
    {script,{"see","1.0"},     %%<label id="boot.tag"/>
     [{preLoaded,preloaded()}, %%<label id="boot.preloaded"/>
      {progress,preloaded},
      {path, [Cwd]},           %%<label id="boot.path"/>
      {primLoad,               %%<label id="boot.preload1"/>
       [lists,
        error_handler,
        see
       ]},                     %%<label id="boot.preload2"/>
      {kernel_load_completed},    %%<label id="boot.kernel"/>
      {progress,kernel_load_completed},
      {progress,started},
      {apply,{see,main,[]}}   %% <label id="boot.apply"/>
     ]},
io:format("Script:~p~n",[Script]),
file:write_file("see.boot", term_to_binary(Script)),
file:write_file("see",[
                       "#!/bin/sh\nerl ",
                       %%" -init_debug ",
                       " -boot ", Cwd, "/see ",
                       "-environment `printenv` -load $1\n"]),
os:cmd("chmod a+x see"),
init:stop(),
true.
```

代码行首先用一个字符串来标注脚本，接下来是一个通过init.erl执行的命令列表。第一个命令（元组{preloaded,[Mods]}）告诉系统已经预加载了哪些模块。接下来是{progress, Atom}命令，progress元组是供调试用的，如果启动Erlang的命令行包含-init_debug标记，就会把它打印出来。

第7行的元组{path, [Dirs]}设置了代码载入器路径，而{primLoad, [Mods]}的意思是载入这个模块列表里的模块。因此，这几行告诉系统要从所给的代码路径里载入三个模块（lists、error_handler和see）。

载入所有代码后，会遇到命令{kernel_load_completed}。它的意思是“已经准备就绪”，接下来会把控制权交给用户代码。在“内核载入完成”这个命令之后（而不是之前）才能调用apply来执行用户函数。最后，第16行里的{apply,{see,main,[]}}会调用apply(see,main,[])。

make_scripts/0必须在Erlang开发环境内执行，因为它调用了code、filename和file模块里的函数，SEE程序是无法使用这些模块的。

现在来构建启动文件和启动脚本。

```
$ erlc see.erl
$ erl -s see make_scripts
Eshell V5.9.3  (abort with ^G)
Script:{script,{"see","1.0"},
         [{preLoaded,[zlib,prim_file,prim_zip,prim_inet,erlang,
                       otp_ring0,init,erl_prim_loader]},
          {progress,preloaded},
          {path,["/Users/joe/projects_active/book/jaerlang2/Book/code/see"]},
          {primLoad,[lists,error_handler,see]},
          {kernel_load_completed},
          {progress,kernel_load_completed},
          {progress,started},
          {apply,{see,main,[]}}]}
```

C.2　在SEE里运行一些测试程序

构建完启动脚本后，就可以把注意力转向将在SEE里运行的程序了。我们列举的所有示例都是普通Erlang模块，它们必须导出main()函数。

程序中最简单的是see_test1。

see/see_test1.erl

```
-module(see_test1).
-export([main/0]).
main() ->
    see:write("HELLO WORLD\n"),
    see:write(integer_to_list(see:modules_loaded()-8) ++ " modules loaded\n").
```

为了运行它，首先要用Erlang开发环境里的标准编译器编译see_test1。做完这步之后，就可以运行程序了。

```
$ erlc see_test1.erl
$ ./see see_test1
Hello world
```

可以给它计时，就像下面这样：

```
$ time ./see see_test1
HELLO WORLD

4 modules loaded
real      0m0.019s
user      0m0.016s
sys       0m0.000s
```

由此可见，载入4个模块（lists、error_handler、see和see_test1）并运行程序总共花了0.019秒。

作为对比，我们可以给OTP系统的对应Erlang程序计时。

see/otp_test1.erl

```
-module(otp_test1).
-export([main/0]).

main() ->
    io:format("HELLO WORLD\n").
```

```
$ erlc otp_test1.erl
$ time erl -noshell -s otp_test1 main -s init stop
HELLO WORLD

real      0m1.127s
user      0m0.100s
sys       0m0.024s
```

用SEE启动并运行我们的小程序比用OTP快59倍。不过请记住，OTP的设计目标并不包括快速启动。对OTP应用程序的期望是启动后能永久运行，所以削减几毫秒的启动时间相对于之后的永久运行来说意义不大。

这里还有一些简单的程序。see_test2负责测试自动载入能否工作。

see/see_test2.erl

```
-module(see_test2).
-export([main/0]).

main() ->
    erlang:display({about_to_call,my_code}),
    2000 = my_code:double(1000),
    see:write("see_test2 worked\n").
```

其中：

see/my_code.erl

```
-module(my_code).
-export([double/1]).

double(X) ->
    2*X.
```

从而：

```
$ ./see see_test2
{about_to_call,my_code}
{new_error_handler,undefined_function,my_code,double,[1000]}
{error_handler,calling,my_code,double,[1000]}
see_test2 worked
```

可以看到`my_code`模块自动载入成功，其余输出内容是一个自定义代码载入器打印出来的调试信息。它是由`error_handler`模块生成的，我们将在本章后面进行讨论。

`see_test3`是一个Erlang程序，它会把标准输入里能看到的所有内容都复制到标准输出里。（这正是Unix管道进程的编写方式。）

see/see_test3.erl

```
-module(see_test3).
-export([main/0]).
-import(see, [read/0, write/1]).

main() -> loop().

loop() ->
    case read() of
        eof ->
            true;
        {ok, X} ->
            write([X]),
            loop()
    end.
```

这里有一个例子：

```
$ cat see.erl | cksum
3915305815 9065
$ cat see.erl | ./see see_test3 see.erl | cksum
3915305815 9065
```

`see_test4`负责测试错误处理。

see/see_test4.erl

```
-module(see_test4).
-export([main/0]).
```

```
main() ->
    see:write("I will crash now\n"),
    1 = 2,
    see:write("This line will not be printed\n").
```

这里有一个例子：

```
$ see see_test4
I will crash now
{stopping_system,{{badmatch,2},
  [{see_test4,main,0,[{file,"see_test4.erl"},{line,6}]}]}}
```

C.3　SEE的API

`see.erl`这个单独模块导出了下列函数。

- main()

  启动系统。
- load_module(Mod)

  载入模块`Mod`。
- log_error(Error)

  在标准输出里打印`Error`的值。
- make_server(Name, FunStart, FunHandler)

  创建一个名为`Name`的永久性服务器。这个服务器的初始状态由`FunStart()`决定，服务器的“处理器”函数是fun`FunHandler`（稍后会作进一步讨论）。
- rpc(Name, Query)

  生成一个对服务器`Name`的远程过程调用`Query`。
- change_behaviour(Name,FunHandler)

  向服务器`Name`发送一个新处理器函数`FunHandler`来改变它的行为。
- keep_alive(Name, Fun)

  确保始终存在一个名为`Name`的注册进程。这个进程由`Fun()`启动（或重启）。
- make_global(Name, Fun)

  生成一个名为`Name`的全局注册进程，它自身会分裂出fun `Fun()`。
- on_exit(Pid, Fun)

  监视`Pid`进程。如果该进程退出并生成原因`{'EXIT', Why}`，就执行`Fun(Why)`。
- on_halt(Fun)

  设置一个条件，使`Fun()`在收到停止系统的请求时执行。如果指定了多个fun就全部调用。
- stop_system(Reason)

  停止系统并生成原因`Reason`。
- every(Pid, Time, Fun)

  只要`Pid`没有终止，就每隔`Time`毫秒执行一次`Fun()`。

- lookup(Key,[{Key,Val}]) -> {found, Val} | not_found

  在某个字典里查找Key键。
- read() -> [string()] | eof

  从标准输入里读取下一行。
- write([string()]) -> ok

  把string写入标准输出。
- env(Name)

  返回环境变量Name的值。

  这些函数可以用于简单的Erlang程序。

C.4　SEE的实现细节

启动一切的shell脚本（see）如下：

```
#!/bin/sh
erl -boot /home/joe/erl/example_programs-2.0/examples-2.0/see\
-environment `printenv` -load $1
```

C.4.1　SEE主程序

系统启动时会执行see:main()，当它终止时，系统也会停止。see:main()的代码如下：

see/see.erl

```
main() ->
    make_server(io,
                fun start_io/0, fun handle_io/2),
    make_server(code,
                const([lists,error_hander,see|preloaded()]),
                fun handle_code/2),
    make_server(error_logger,
                const(0),  fun handle_error_logger/2),
    make_server(halt_demon,
                const([]), fun handle_halt_demon/2),
    make_server(env,
                fun start_env/0, fun handle_env/2),
    Mod = get_module_name(),
    load_module(Mod),
    run(Mod).
```

它会启动5个服务器（io、code……），载入错误处理器，找出待运行模块的名称并载入该模块，然后运行模块里的代码。run(Mod)会分裂并连接Mod:main()，然后等待它终止。当它终止时会调用stop_system。

see/see.erl

```
run(Mod) ->
    Pid = spawn_link(Mod, main, []),
    on_exit(Pid, fun(Why) -> stop_system(Why) end).
```

main/0会启动许多不同的服务器。在介绍这些服务器的工作细节之前，先来看看用于构建客户端–服务器的通用框架。

C.4.2　SEE里的客户端–服务器模型

要创建一个服务器，我们会调用make_server(Name, Fun1, Fun2)。Name是服务器的全局名称，Fun1()的预期返回值是State1，也就是服务器的初始状态。如果是远程过程调用Fun2(State, Query)，这个函数就应当返回{Reply, State1}；如果是cast调用，就只返回State1。make_server的代码如下：

see/see.erl

```
make_server(Name, FunD, FunH) ->
    make_global(Name,
                fun() ->
                        Data = FunD(),
                        server_loop(Name, Data, FunH)
               end).
```

这个服务器的循环就像下面这样：

see/see.erl

```
server_loop(Name, Data, Fun) ->
    receive
        {rpc, Pid, Q} ->
            case (catch Fun(Q, Data)) of
                {'EXIT', Why} ->
                    Pid ! {Name, exit, Why},
                    server_loop(Name, Data, Fun);
                {Reply, Data1} ->
                    Pid ! {Name, Reply},
                    server_loop(Name, Data1, Fun)
            end;
        {cast, Pid, Q} ->
            case (catch Fun(Q, Data)) of
                {'EXIT', Why} ->
                    exit(Pid, Why),
                    server_loop(Name, Data, Fun);
                Data1 ->
                    server_loop(Name, Data1, Fun)
            end;
        {eval, Fun1} ->
            server_loop(Name, Data, Fun1)
    end.
```

要对服务器进行查询，会使用rpc（Remote Procedure Call的缩写，即远程过程调用），它的代码如下：

see/see.erl

```
rpc(Name, Q) ->
    Name ! {rpc, self(), Q},
    receive
        {Name, Reply} ->
            Reply;
        {Name, exit, Why} ->
            exit(Why)
    end.
```

请注意服务器循环里的代码是如何与rpc交互的。服务器里的处理函数fun受catch保护，如果服务器抛出一个异常错误，就会把{Name,exit,Why}消息发回客户端。如果客户端收到了这个消息，就会执行exit(Why)来抛出一个异常错误。

这种机制的最终效果就是让客户端抛出一个异常错误。请注意，如果服务器无法处理客户端发送的查询，就会以原有状态继续运行。

所以，对服务器而言，远程过程调用就像事务一样。如果它们不能完整执行，服务器就会回滚到远程过程调用发起之前的状态。

如果只想给服务器发一个消息而不关心它是否回复，就调用cast/2。

see/see.erl

```
cast(Name, Q) ->
    Name ! {cast, self(), Q}.
```

可以给服务器发送另一个用于循环的fun，从而改变服务器的行为。

see/see.erl

```
change_behaviour(Name, Fun) ->
    Name ! {eval, Fun}.
```

别忘了，启动服务器时的初始数据结构经常是一个常量。可以定义const(C)，让它返回一个函数，此函数执行后会返回C。

see/see.erl

```
const(C) -> fun() -> C end.
```

现在再把注意力放到各种服务器上。

C.4.3　代码服务器

代码服务器的启动方式是执行下面这个函数：

see/see.erl

```
make_server(Name, FunD, FunH) ->
    make_global(Name,
                fun() ->
                       Data = FunD(),
                       server_loop(Name, Data, FunH)
               end).
```

load_module(Mod)会对代码服务器进行远程过程调用。

```
load_module(Mod) ->
    rpc(code, {load, Mod}).
```

代码服务器的全局状态是[Mod]，也就是一个包含所有已加载模块的列表。（它的初始值是[init, erl_prim_loader]。这些模块是预加载的，被编译到Erlang运行时系统的内核里。）

服务器的处理函数handle_code/see/2如下：

```
handle_code(modules_loaded, Mods) ->
    {length(Mods), Mods};
handle_code({load, Mod}, Mods) ->
    case member(Mod, Mods) of
        true ->
            {already_loaded, Mods};
        false ->
            case primLoad(Mod) of
                {ok,Mod} ->
                    {{ok,Mod}, [Mod|Mods]};
                Error ->
                    {Error, Mods}
            end
    end.
```

primLoad负责载入工作：

```
primLoad(Module) ->
    Str = atom_to_list(Module),
    case erl_prim_loader:get_file(Str ++ ".beam") of
        {ok, Bin, _FullName} ->
            case erlang:load_module(Module, Bin) of
                {module, Module} ->
                    {ok,Module};
                {module, _} ->
                    {error, wrong_module_in_binary};
                _Other ->
                    {error, {bad_object_code, Module}}
            end;
        _Error ->
            {error, {cannot_locate, Module}}
    end.
```

C.4.4　错误记录器

log_error(What)会记录标准输出上的错误What，它是以播发的形式实现的。

see/see.erl

```
log_error(Error)  ->  cast(error_logger, {log, Error}).
```

对应的服务器处理函数如下：

see/see.erl

```
handle_error_logger({log, Error}, N) ->
    erlang:display({error, Error}),
    {ok, N+1}.
```

请注意，错误记录器的全局状态是一个整数N，代表已发生错误的总数。

C.4.5　停止守护程序

停止守护程序（halt demon）是在系统停止时调用的。执行on_halt(Fun)会设置一个条件，使Fun()在系统停止时执行。停止系统的方法是调用stop_system()函数。

see/see.erl

```
on_halt(Fun)     -> cast(halt_demon,{on_halt,Fun}).
stop_system(Why) -> cast(halt_demon,{stop_system,Why}).
```

它的服务器处理代码如下：

see/see.erl

```
handle_halt_demon({on_halt, Fun}, Funs) ->
    {ok, [Fun|Funs]};
handle_halt_demon({stop_system, Why}, Funs) ->
    case Why of
        normal -> true;
        _      -> erlang:display({stopping_system,Why})
    end,
    map(fun(F) -> F() end, Funs),
    erlang:halt(),
    {ok, []}.
```

C.4.6　I/O服务器

I/O服务器允许对STDIO进行访问。read()会从标准输入里读取一行，write(String)则会把一个字符串写入标准输出。

see/see.erl

```
read()   -> rpc(io, read).
write(X) -> rpc(io, {write, X}).
```

I/O服务器的初始状态通过执行`start_io()`获得。

see/see.erl

```
start_io() ->
    Port = open_port({fd,0,1}, [eof, binary]),
    process_flag(trap_exit, true),
    {false, Port}.
```

I/O处理器的代码如下：

see/see.erl

```
handle_io(read, {true, Port}) ->
    {eof, {true, Port}};
handle_io(read, {false, Port}) ->
    receive
        {Port, {data, Bytes}} ->
            {{ok, Bytes}, {false, Port}};
        {Port, eof} ->
            {eof, {true,Port}};
        {'EXIT', Port, badsig} ->
            handle_io(read, {false, Port});
        {'EXIT', Port, _Why} ->
            {eof, {true, Port}}
    end;
handle_io({write,X}, {Flag,Port}) ->
    Port ! {self(), {command, X}},
    {ok, {Flag, Port}}.
```

I/O服务器的状态是`{Flag, Port}`。如果遇到`eof`，`Flag`就是`true`，否则就是`false`。

C.4.7　环境服务器

`env(E)`函数的作用是找出环境变量E的值。

see/see.erl

```
env(Key) -> rpc(env, {lookup, Key}).
```

这个服务器的代码如下：

```
handle_env({lookup, Key}, Dict) ->
    {lookup(Key, Dict), Dict}.
```

服务器的初始值通过执行以下代码获得：

```
start_env() ->
    Env = case init:get_argument(environment) of
              {ok, [L]} ->
                  L;
```

```
                error ->
                    fatal({missing, '-environment ...'})
        end,
    map(fun split_env/1, Env).
split_env(Str) -> split_env(Str, []).
```

C.4.8　全局进程支持函数

我们需要一些方法来保持进程运行和注册进程的全局名称。

keep_alive(name, Fun)会生成一个名为Name的注册进程。它是由Fun()启动的，如果进程挂了，就会被自动重启。

see/see.erl

```
keep_alive(Name, Fun) ->
    Pid = make_global(Name, Fun),
    on_exit(Pid,
            fun(_Exit) -> keep_alive(Name, Fun) end).
```

make_global(Name, Fun)会检查是否存在一个注册名为Name的全局进程。如果不存在，它就会分裂出一个进程来执行Fun()，并注册进程名Name。

see/see.erl

```
make_global(Name, Fun) ->
    case whereis(Name) of
        undefined ->
            Self = self(),
            Pid = spawn(fun() ->
                                make_global(Self,Name,Fun)
                        end),
            receive
                {Pid, ack} ->
                    Pid
            end;
        Pid ->
            Pid
    end.
make_global(Pid, Name, Fun) ->
    case register(Name, self()) of
        {'EXIT', _} ->
            Pid ! {self(), ack};
        _ ->
            Pid ! {self(), ack},
            Fun()
    end.
```

C.4.9　进程支持函数

on_exit(Pid, Fun)与Pid相连。如果Pid崩溃并生成原因Why，就会执行Fun(Why)。

see/see.erl

```
on_exit(Pid, Fun) ->
    spawn(fun() ->
                  process_flag(trap_exit, true),
                  link(Pid),
                  receive
                      {'EXIT', Pid, Why} ->
                          Fun(Why)
                  end
          end).
```

every(Pid, Time, Fun)与Pid相连，Fun()会每隔Time时间执行一次。如果Pid退出，这个过程就会停止。

see/see.erl

```
every(Pid, Time, Fun) ->
    spawn(fun() ->
                  process_flag(trap_exit, true),
                  link(Pid),
                  every_loop(Pid, Time, Fun)
          end).

every_loop(Pid, Time, Fun) ->
    receive
        {'EXIT', Pid, _Why} ->
            true
    after Time ->
            Fun(),
            every_loop(Pid, Time, Fun)
    end.
```

C.4.10　工具函数

get_module_name()会从命令行获取模块名。

see/see.erl

```
get_module_name() ->
    case init:get_argument(load) of
        {ok, [[Arg]]} ->
            module_name(Arg);
        error ->
            fatal({missing, '-load Mod'})
    end.
```

C.5　Erlang如何载入代码

Erlang的默认代码载入机制是“按需”载入代码。代码被首次调用时，如果系统发现代码缺失，就会载入它。

来看看具体的过程。假设程序调用了`my_mod:myfunc(Arg1, Arg2, ... ArgN)`函数，但这个模块的代码尚未被载入，系统就会自动把这个调用转换成下面的形式：

```
error_module:undefined_function(mymod, myfunc, [Arg1, Arg2, ..., ArgN])
```

`undefined_function`类似下面这样：

```
undefined_function(Mod, Func, ArgList) ->
     case code:load_module(Mod) of
         {ok, Bin} ->
             erlang:load_module(Mod, Bin),
             apply(Mod, Func, ArgList);
         {error, _ } ->
             ...
     end
```

这个未定义函数处理器把寻找模块代码的任务指派给代码处理器。如果代码处理器能找到目标代码，就会载入模块然后调用`apply(Mod, Func, ArgList)`，系统则会继续运行，就像调用转换没有发生过一样。

负责此事的SEE代码遵循这个模式：

see/error_handler.erl

```
-module(error_handler).
-export([undefined_function/3,undefined_global_name/2]).
undefined_function(see, F, A) ->
    erlang:display({error_handler,undefined_function,
                    see,F,A}),
    exit(oops);
undefined_function(M, F, A) ->
    erlang:display({new_error_handler,undefined_function,M,F,A}),
    case see:load_module(M) of
        {ok, M} ->
            case erlang:function_exported(M,F,length(A)) of
                true ->
                    erlang:display({error_handler,calling,M,F,A}),
                    apply(M, F, A);
                false ->
                    see:stop_system({undef,{M,F,A}})
            end;
        {ok, _Other} ->
            see:stop_system({undef,{M,F,A}});
```

```
        already_loaded ->
            see:stop_system({undef,{M,F,A}});
        {error, What} ->
            see:stop_system({load,error,What})
    end.
undefined_global_name(Name, Message) ->
    exit({badarg,{Name,Message}}).
```

注意　Erlang系统的错误处理器和这里的代码存在显著区别，它所做的远远多于这个简单的代码处理器。它必须跟踪模块版本和其他一些让人费心的对象。

错误处理器被启动脚本载入后，自动载入功能就能正常工作了。它使用SEE里的代码处理机制，而不是OTP系统的机制。

C.6　练习

(1) SEE提供了自动载入功能，但我们可以编写一个更简单的（不带自动载入）。移除与自动载入有关的代码。

(2) 你甚至不需要SEE就能编写完全独立的应用程序。编写一个最简单的程序，让它把“Hello world”写入标准输出然后终止。

(3) 编写一个最简单的cat程序，让它把标准输入里的内容复制到标准输出（就像see_test3.erl一样），但不通过SEE启动。复制一些大文件并计时，然后与一些流行的脚本语言进行比较。